THE UNIFRAME QUALITY OF SERVICE FRAMEWORK

A Thesis

Submitted to the Faculty

of

Purdue University

by

Girish Jagadeeshwaraiah Brahnmath

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2002

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
DEC 2002		**00-00-2002 to 00-00-2002**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
The Uniframe Quality of Service Framework	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Indiana University/Purdue University,Department of Computer and Information Sciences,Indianapolis,IN,46202**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**144**	

To Ma and Pa.

ACKNOWLEDGMENTS

Graduate studies at the Department of Computer and Information Science at Indiana University-Purdue University have been an enriching experience to me. It helped me acquire skills essential to be a good computer professional. My work as a Research Assistant on the UniFrame project was a great opportunity to further enhance the skills acquired during my Graduate studies. I would like to take this opportunity to express my gratitude to all those who made this thesis possible.

I would like to profoundly thank my advisor Dr. Rajeev Raje, for giving me the opportunity to be a part of the UniFrame project and for guiding me throughout my graduate studies and research work. His inputs and insight were invaluable in producing this thesis. I am grateful to him for his constant encouragement, making me reach higher and farther in all my academic endeavors.

I wish to thank Dr. Andrew Olson for providing me with valuable advice and input during the course of my academic and research work. I would especially like to thank him for his valuable feedback on my thesis.

I would also like to thank Dr. Stanley Chien for being on my thesis committee and for reviewing my thesis.

I am thankful to the U.S. Department of Defense and the U.S. Office of Naval Research for supporting this research with their grant under the award number N00014-01-1-0746.

I would like to thank all my teammates on the UniFrame project, the faculty and staff of the Computer Science Department for their co-operation and assistance towards this thesis.

Finally, I would like to thank my parents, my brother and sister-in-law for their kind love, encouragement and support.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Brahnmath, Girish Jagadeeshwaraiah, M.S., Purdue University, December 2002. The UniFrame Quality of Service Framework. Major Professor: Rajeev Raje.

The Component-based Software Development (CBSD) is now being recognized as the direction towards which the software industry is headed. In order for this approach to result in software systems with predictable quality, the components utilized to build software systems should offer a guaranteed level of quality. However, there is a lack of standardization within the software community regarding the quality of software components. Also, according to the CBSD philosophy, a given component may be used under diverse operating environments and usage patterns, which can affect the Quality of Service (QoS) offered by the software component. This calls for an objective paradigm for quantifying and specifying the quality of software components, as well as, accounting for the effects of the environment and the effects of usage patterns on the QoS of software components. This thesis presents a QoS framework, called the UniFrame Quality of Service (UQOS) framework created as a part of the UniFrame Project, to address the above mentioned issues. The UQOS framework consists of four major parts namely, the QoS Catalog, the approach for accounting for the effects of environment on the QoS of software components, the approach for accounting for the effects of usage patterns on the QoS of software components and the specification of the QoS of software components. The QoS catalog is intended to act as a tool for standardizing the notion of Quality of software components. The approaches to account for the effects of the environment and the effects of usage patterns on the QoS of components consist of an empirical validation of the QoS of software components under diverse environmental conditions and usage patterns, and specification of the resulting QoS values in the component interface. These experiments and their results are presented and analyzed.

1. INTRODUCTION

The world of computer software has constantly evolved from its infancy towards a state of maturity. There has been a constant endeavor on the part of computer scientists to bring Computer Science on par with its more mature peers like the physical sciences. The emergence of component-based software development is a concrete step in this direction.

For many years, the software development had consisted mainly of custom made software built individually for specific clients. With the advent of Object-Oriented Programming the concept of code reuse became a highly popular cost-effective programming technique.

Component-Based Software Development (CBSD) is taking this a step further by developing entire software systems by selecting appropriate Commercial off the shelf (COTS) software components. [SZY99] defines a software component as "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". At the same time, the advent of high speed networks combined with the growing popularity of the Internet has resulted in a paradigm shift in the software industry towards distributed computing. Thus, the popularity of component-based distributed software systems can be seen as a natural outcome of the combination of the above two phenomena. However, there are few issues that need to be addressed in order for the development of component-based distributed software to gain support from the software community.

One of these issues is the existence of numerous diverse distributed computing models (like J2EE™, .NET™, CORBA™, etc) in the software community. Some of these models have proved to be quite popular among the academic and industrial circles.

This has resulted in a situation where several different distributed computing models are forced to co-exist. However, these models mostly do not provide sufficient facilities to interact with each other seamlessly. The interoperability which they provide is limited mainly to the underlying hardware, operating system and/or implementation languages.

If component-based distributed software systems are to become successful, then there is certainly a need for an approach that will transcend this limited interoperability. One possible approach to achieve comprehensive interoperability is that of using a meta-model for heterogeneous distributed components. Web Services [WES02] are viewed as a possible solution to this problem. [MAY02] defines Web Services as "Web Services are a standards-based software technology that lets programmers and integrators combine existing and new systems or applications in new ways over the Internet, within a company's boundaries, or across many companies. Web Services allow interoperability between software written in different programming languages, developed by different vendors, or running on different Operating Systems or platforms".

The other issue is regarding the quality of the COTS Components used in CBSD. [ISO86] defines QoS as "The totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs". In order for the CBSD approach to result in software systems with a predictable quality, the COTS components utilized, should in turn offer a guaranteed level of quality. However, currently there are no standardized frameworks that incorporate Quality of Service (QoS) as an inherent part of software components. This can lead to inconsistencies and irregularities in the representation of a component's quality. This calls for a concrete framework which incorporates QoS as an inherent part of software components and offers objective means to quantify, verify, validate and specify the QoS of software components.

The UniFrame and UniFrame Approach (UA) [RAJ01, RAJ02] provide a framework that allows a seamless interoperation of heterogeneous and distributed software components and incorporates the following key concepts: a) a meta-component model (the Unified Meta Model – UMM [RAJ00]), with a associated hierarchical setup for indicating the contracts and constraints of the components and associated queries for

integrating a distributed system, b) an integration of the QoS at the individual component and distributed application levels, c) the validation and assurance of the QoS, based on the concept of event grammars, and e) generative rules, along with their formal specifications, for assembling an ensemble of components out of available component choices. The UniFrame Quality of Service (UQOS) Framework, which is the topic of this thesis, is an implementation of the QoS aspects of the UniFrame Approach.

It is believed that the UniFrame approach, along with the associated UQOS framework, provide the necessary solutions to the issues identified earlier, which affect the development of component-based distributed software systems.

1.1. Problem Definition and Motivation

CBSD involves usage of appropriate COTS software components towards creation of software systems. The notion of assembling complete systems out of prefabricated parts is prevalent in many branches of science and engineering such as manufacturing. This leads to the creation of prompt and economical products. This is possible because of the existence of standardized components that meet a manufacturer's functional and non-functional (quality) requirements. Also, the task of the manufacturer is made much easier because of the presence of standardized component catalogs outlining their functional and non-functional parameters.

At present, a software developer who uses component-based approach cannot enjoy the same luxury. This is mainly because most COTS components are specified only with functional parameters in their interfaces. Typically, no concrete notion of quality is associated with components. Hence, the system developer has no means to objectively compare the performance characteristics of multiple components with the same functionality. This tends to restrict the developer's options when trying to select a component with a given functionality during the software development process, especially, in quality-critical applications. Thus, there is a need for a framework that would allow objective measurements of a component's QoS parameters. The creation of a Quality of Service catalog for software components would be the first step in this

direction. Such a catalog should contain detailed descriptions about QoS parameters of software components along with the appropriate metrics, evaluation methodologies and the interrelationships with other parameters.

According to the CBSD philosophy, a given component may be used under diverse environments. The definition of environment here includes those features (called *environment variables*) of the execution platform of a software component, which might have a significant impact on the QoS of that component. Some of these environment variables are, the CPU speed, the memory, the process priority assigned to the execution of a component and the operating system used. The fact that the environment variables can affect the QoS of a software component implies that any QoS value associated with a software component would not necessarily hold true in foreign environments. Hence, it becomes critical to account for the effect of the execution environment on the QoS of software components.

Also, once a component is deployed on the network by the component user, it may be subjected to varying usage patterns. For instance, an e-commerce component, once deployed on the Internet, may be subject to varying number of users and user requests depending on factors like, the time of the day, the time of the year (seasonal variation), the deployment site and the semantics of the application. The variations in the pattern of users and user requests (the usage patterns) can have a profound impact on the QoS of a component (and in turn, the level of satisfaction of the end-user or consumer). This in effect implies that it is crucial to be able to deduce the effect of usage patterns on the QoS of software components.

The UQOS framework is intended to address the issues raised here, about the quality of software components.

1.2. Objectives: Statement of Goals

The specific objectives of this thesis are:

- To provide a framework to objectively quantify the QoS or non-functional parameters of software components and to make QoS parameters an inherent

part of software components. This objective is composed of the following sub-objectives.

- To create a Quality of Service Catalog to standardize the notion of quality of software components and to act as a reference guide for software component developers (producers) and system integrators (consumers).

- To investigate the effect of environment on the QoS of software components and provide a mechanism to incorporate the effect of environment on QoS into the component development process.

- To study the effect of usage patterns on the QoS of software components and provide a mechanism to incorporate the effect of usage patterns on QoS into the component development process.

- To investigate the various existing QoS specification schemes, and adopt the scheme most compatible with the UniFrame Approach and its objectives.

The approach used in this thesis to achieve the above-mentioned objectives is as follows:

- The Creation of a QoS Catalog with the intention to act as a tool for standardizing the notion of the Quality of software components. The catalog contains detailed descriptions about QoS parameters of software components, including the metrics, the evaluation methodologies, the factors influencing these parameters and the interrelationships among these parameters.
- The proposal of standard approaches to account for the effect of the environment and the effect of usage patterns on the QoS of software components. These approaches involve an empirical evaluation of the effect of environment and the effect of usage patterns on the values of the QoS

parameters of a component, and a specification of the resulting QoS values in the component interface.

- The incorporation of the Component Quality Modeling Language (CQML) [AAG01] into the UQOS framework for specifying the QoS of software components.

1.3. Contributions of this thesis

The contributions of this thesis are as follows:

- It provides a QoS framework for the UniFrame Approach, and possible solutions to some of the QoS-related issues affecting component-based software development, as described in the section 1.1.

- It provides a QoS Catalog for software components containing
  - A compilation of commonly used QoS parameters, along with their definitions.
  - A classification of these parameters based on criteria like domain of usage, static or dynamic behavior, nature of the parameters and the composability of the parameters.
  - An incorporation of methodologies for quantifying the QoS parameters.
  - The set of factors influencing each of the identified QoS parameters.
  - The interrelationships between the QoS parameters.

- Proposes an approach to account for the effect of environment (as described in section 1.1) on the QoS of software components.

- Proposes an approach to account for the effect of usage patterns (as described in section 1.1) on the QoS of software components.

- Provides a case-study from the math domain to illustrate the applicability of the proposed solution in a real-world scenario.

1.4. Organization of this thesis

The thesis is organized into six chapters. Chapter 1 provides an introduction to the thesis, along with the problem definition and motivation, objectives, and contributions of the thesis. Chapter 2 presents a survey of other related approaches to QoS across the different levels of a distributed system and a perspective of the UQOS framework in relation to the other related work. Chapter 3 describes the details of the UniFrame Approach and associated Unified Meta Model (UMM), followed by a discussion of the role of the UQOS framework in the UniFrame Approach and the objectives of the UQOS framework. Chapter 4 provides a detailed description of the implementation of the UQOS framework, consisting of four parts, namely, the QoS Catalog for software components, the approach for accounting for the effects of environment on the QoS of software components, the approach for accounting for the effects of usage patterns on the QoS of software components and the specification of the QoS of software components. In Chapter 5, a case-study from the math domain is provided to illustrate the applicability of the UQOS framework in a real-world scenario. Chapter 6 provides a conclusion to the thesis by listing the features of the UQOS framework, possible enhancements to the framework as future work and a summation of the thesis.

2. BACKGROUND AND RELATED WORK

In the previous chapter, a brief introduction to this thesis involving the UQOS framework was presented, along with the problem definition and motivation, objectives, and contributions of the thesis. In this chapter, the work related to QoS at different levels of a distributed computing system is presented, along with the details of the features that distinguish UQOS from the other related works.

Quality of Service (QoS) is now well recognized in all fields of science and technology as a reflection of product performance and reliability. Over the years there have been several efforts made to incorporate QoS into computer hardware and software. These efforts initially started out in the field of networking and slowly spread out into various other disciplines of computer science [FER98].

To simplify the task of reviewing the related work, various efforts have been broadly classified into five categories, namely: QoS in Networks, QoS in Operating Systems, QoS in Middleware, QoS in Applications and QoS in Software Components. These five categories can be broadly viewed as being organized in a hierarchical fashion with the QoS in Networks being on the bottom-most tier, the QoS in Operating Systems being on the next higher or second tier, the QoS in middleware being on the third tier and the QoS in Applications and Software Components being on the topmost tier. Presented below is an overview of some of the significant work in each of the above mentioned categories. This is followed by an overview of the distinguishing features of the UQOS framework, which can be categorized as QoS in software components.

2.1. QoS in Networks

The notion of Quality of Service has been largely associated with the field of networking. There are several QoS mechanisms in existence for data networks and some of the significant among these are presented below.

2.1.1. Asynchronous Transfer Mode (ATM)

Asynchronous Transfer Mode (ATM) is the most widely deployed backbone technology in the world. [GAR97, KUR01] describe the following five quality of service categories within the ATM:

i. Constant bit rate (CBR) Service: It is defined as a simple, reliable and guaranteed channel. In ATM, the ATM cells are the basic units of transmission and are analogous to packets. In CBR service, ATM packets are transmitted across the network in such a fashion as to ensure that the end-to-end delay experienced by a cell, the variability in the end-to-end delay (jitter) and the fraction of cells that are lost or delivered late, are guaranteed to be within specified limits. Also, an allocated transmission rate for a given connection is pre-defined by the sender, and the sender is assumed to offer traffic to the connection constantly at this rate. CBR service is well-suited for transmission of real-time, constant-bit-rate audio (for example, a digitized telephone call) and video traffic.

ii. Unspecified bit rate (UBR) Service: UBR is a service without any explicit rate parameter. It is considered a generic best-effort service. There is no cell-wise delay or jitter requirement, nor any explicit loss rate contract. The QoS in this case is determined by engineering the capacity of the network to accommodate the overall traffic demands and not by algorithms operating on each cell. Unlike CBR service, UBR service makes no guarantees with respect to rate, delay, jitter, and loss, other than in-order delivery of cells. It is thus equivalent to the Internet's

best-effort service model. It is well suited for non-interactive data transfer applications like email and newsgroups.

iii. Non-real-time variable bit rate (nrt-VBR) Service: To improve the loss and delay that might be encountered with UBR, the nrt-VBR was established. It provides peak and sustainable rate parameters, as well as a loss rate parameter. These are used to allocate resources for each nrt-VBR connection. The loss rate allows the service category to be engineered for statistical multiplexing while maintaining acceptable performance.

iv. Real-time variable bit rate (rt-VBR) Service: In this service category, the source transmission rate is allowed to vary according to parameters specified by the user of the network. The acceptable cell loss rate, delay and jitter are explicitly specified. It was established to accommodate audio, video and other data traffic that is generated with a variable bit rate.

v. Available bit rate (ABR) Service: The ABR service is considered a “better” best-effort service. It offers a minimum cell transmission rate to a connection and in case, the network has enough free resources at a given time, it allows a sender to transmit at a higher rate than the minimum cell transmission rate. Thus, ABR provides a minimum bandwidth guarantee and attempts to transfer data as fast as possible. Hence, it is well suited for applications that require low transfer delays, like web browsing.

The ATM service models provide varying levels of QoS guarantees in data networks as described above. The user can adopt a particular service model depending on the nature of his/her application.

2.1.2. Integrated Services (IntServ)

Integrated Services is a set of standards set down by IETF (Internet Engineering Task Force) [SHE97]. It is a framework for defining services in which multiple classes of traffic can be assured of different QoS profiles by the network elements. Here, the applications must have the knowledge of the characteristics of their traffic *a priori* and signal the intermediate network elements to reserve certain resources to meet its (the application's) traffic properties. The integrated services model [CLA94] proposes two additional service classes on top of best-effort service, namely:

i. Guaranteed Service which is applicable to applications that require a fixed bound on delay and

ii. Controlled Load Service for applications that demand reliable and enhanced best-effort service.

IntServ is typically used in association with the Resource Reservation Protocol, described in section 2.1.2.1, to provide individualized QoS guarantees to individual application sessions.

2.1.2.1. Resource Reservation Protocol (RSVP)

RSVP is a signaling protocol designed for applications that need to reserve resources. "RSVP protocol is used by a host, on behalf of an application data stream, to request a specific quality of service from the network for particular data streams or flows. The RSVP protocol is also used by routers to deliver QoS control requests to all nodes along the path(s) of the flows and to establish and maintain state to provide the requested service." [ZHA96].

RSVP is not designed to act as a routing protocol; it operates along with separate unicast and multicast protocols. RSVP treats the sender and receiver as distinct entities and it requests resources in only one direction. It occupies the slot of the transport protocol and operates on top of Internet Protocol (IP). RSVP provides the signaling

mechanism to reserve per-flow resources at routers within the network and facilitates in providing guaranteed QoS.

2.1.3. Differentiated Services (DiffServ)

In the DiffServ model, the network traffic is classified and conditioned at the entry to a network and assigned to different behavior aggregates [BLA98]. DiffServ defines a field in packets' IP headers, called the DiffServ code point (DSCP). Hosts or routers sending traffic into a DiffServ network mark each transmitted packet with a DSCP value. Routers within the DiffServ network use the DSCP to classify packets and apply specific queuing behavior based on the results of the classification. Traffic from different flows having similar QoS requirements is marked with the same DSCP, thus aggregating the flows to a common queue or scheduling behavior.

[HEI99] defines different classes of services that could be implemented using DiffServ. A set of services called Olympic Services is described, which consist of three service classes namely, bronze, silver, and gold. Packets may be assigned to any one of these classes. The packets in the gold class experience lighter load and hence, have greater probability for timely forwarding than packets assigned to the silver class. Similarly, packets belonging to silver class experience lesser load than packets belonging to the bronze class. Also, it is possible to segregate packets within each class by giving them either low, medium, or high drop precedence.

In this section, the major QoS mechanisms in computer networks were briefly described. It is to be noted that these mechanisms operate independently of each other, but could be used simultaneously. Also, these mechanisms interact directly with the underlying network hardware to provide QoS guarantees through resource reservation, packet classification, isolation of traffic flows, scheduling and policing (regulating the rate at which a flow can inject packets into the network). The majority of the higher-level mechanisms, which are described in the later sections, including the UQOS framework, utilize these network level mechanisms to realize their QoS guarantees.

Hence, the QoS mechanisms in networks provide the foundation for implementing the QoS guarantees in distributed computing systems.

2.2. QoS in Operating Systems

The QoS mechanisms in the networking world ensure that the network connecting the end nodes delivers the required QoS by using concepts such as resource reservation. Along similar lines, the QoS mechanisms for operating systems provide the required QoS at the end nodes (individual systems) by reserving the resources local to the system. A review of some of the major QoS enabled operating systems is given below.

2.2.1. Eclipse Operating System

Eclipse is derived from the Plan9 operating system from Bell labs [PIK95]. It provides the reservation and scheduling of CPU, I/O and physical memory. The resources are managed independently using the shell, without using system-level programming.

Eclipse utilizes a new operating system abstraction called reservation domains. A reservation domain is a collection of processes and corresponding resource reservations. Here, each reservation domain is assigned a certain percentage of each resource like 40% disk I/O, 50% CPU, etc. Hence, each reservation domain acts like a small dedicated machine. The processes that belong to a particular domain are guaranteed to receive at least their portions of the domains' associated resources by means of resource reservation. The sharing of resources is implemented using the concept of locks, where a process holding a lock on a resource has the access rights to the resource. Reservation domains enable explicit control over the provisioning of system resources among applications to achieve the desired levels of predictable performance.

Eclipse also uses a new scheduling algorithm called Move-to-rear List Scheduling (MTR-LS), which provides a cumulative service guarantee, in addition to bounds on fairness and delay [BRU98]. MTR-LS uses an ordered list of active processes called 'L' and a constant 'T' called virtual time quantum. Each process P_i in L is associated with a

value $left_i$ called the size of the quantum, which is the maximal amount of service time process P_i can receive without interruption. The initial value of $left_i$ is equal to $\alpha_i T$ (greater than 0). After a given process is serviced, $left_i$ is decremented by the process service time. If the result is zero, then $left_i$ is moved to the rear of the list L and value of $left_i$ is reset to $\alpha_i T$.

Eclipse thus utilizes the concepts of reservation domains and move-to-rear scheduling policy as the mechanisms to implement QoS based handling of processes.

2.2.2. '2K' Operating System

[KON00] defines 2K as an integrated operating system architecture that addresses the problems of resource management in heterogeneous networks, dynamic adaptability, and configuration of component-based distributed applications. It runs on top of existing operating systems like Linux™, Solaris™ and Windows™.

A network-centric model is adopted in 2K, in which all entities exist on the network, represented as CORBA objects. Each entity is characterized by a network-wide identity, network-wide profile and dependencies upon other network entities. Here, the entities that constitute a service are assembled when that particular service is instantiated.

The system philosophy is to configure an application automatically and load a minimal set of components required for the most efficient execution of the application. This philosophy is achieved by utilizing standard CORBA services like the Event Manager Service and the Trader Service and extending it with the addition of services like the Automatic Configuration Service and QoS Aware Resource Management Service. The details of these are provided later in this section.

The 2K Automatic Configuration Service manages two distinct kinds of dependencies namely:

- Prerequisites specified by the user, which consist of any special requirements for properly loading, configuring and executing a component, such as the type and

share of hardware resources that a component needs and the other software components that it requires.

- Dynamic dependencies among loaded components in a running system. With information regarding their runtime dependencies, the applications can select different components to fulfill their needs in different environments.

Automatic configuration service is responsible for the automatic assembly of applications. It parses the prerequisites and checks if it is necessary to create new instances of the required components. If so, it fetches the required components from a component repository and dynamically loads them. Thus, only a minimal number of required components are loaded at run time. During the process of automatic configuration, the automatic configuration service creates a runtime representation of inter-component dependencies using CORBA objects called ComponentConfigurators. The ComponentConfigurators contain lists of CORBA Interoperable Object References (IORs) which point to other components and ComponentConfigurators, leading to a dependence graph of distributed components. It is possible for applications to implement specialized instances of component configurators to adapt to variation in CPU load and resource availability. [KON01] illustrates the Automatic Configuration framework as shown in figure 2.1 (reproduced from [KON01] with permission).

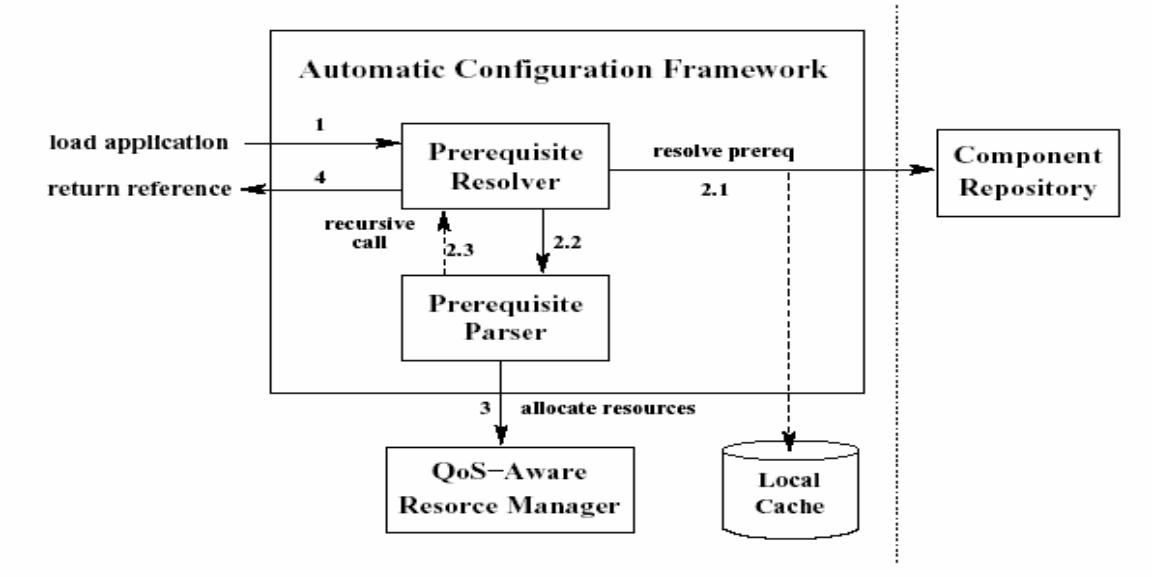

Figure 2.1. Automatic Configuration Service in the 2K Operating System

The QoS-Aware resource management in 2K relies on Local Resource Managers (LRMs) present in each node of a 2K cluster. The LRMs export the hardware resources in each node to the whole distributed system. LRMs also provide periodic updates of the state of the resources under them to the Global Resource Manager (GRM). GRM is a replicated service that maintains an approximate view of the 2K cluster resource utilization. The GRM utilizes the information sent by LRMs to perform QoS-aware load distribution within its cluster. The LRMs also handle the tasks of QoS-aware admission control, resource negotiation, reservation and scheduling of tasks within a single node. The QoS-Aware resource management in the 2k Operating System as illustrated in [KON01] is shown in figure 2.2 (reproduced from [KON01] with permission).

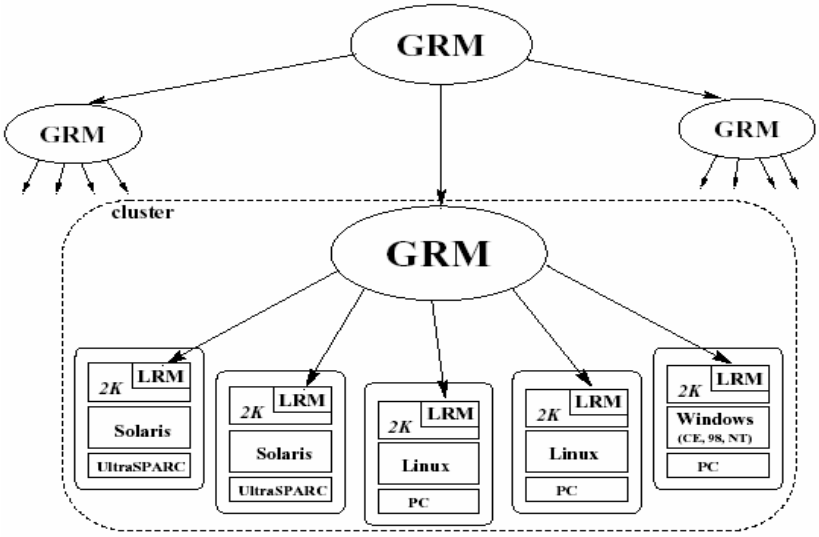

Figure 2.2. QoS-Aware resource management in the 2k Operating System

A CORBA Trader supplies 2K with resource discovery services. This allows applications to request resources based on QoS specifications. The architectural framework of 2K as illustrated in [KON01] is shown in figure 2.3 (reproduced from [KON01] with permission). It indicates the interactions between the various services described earlier in this section.

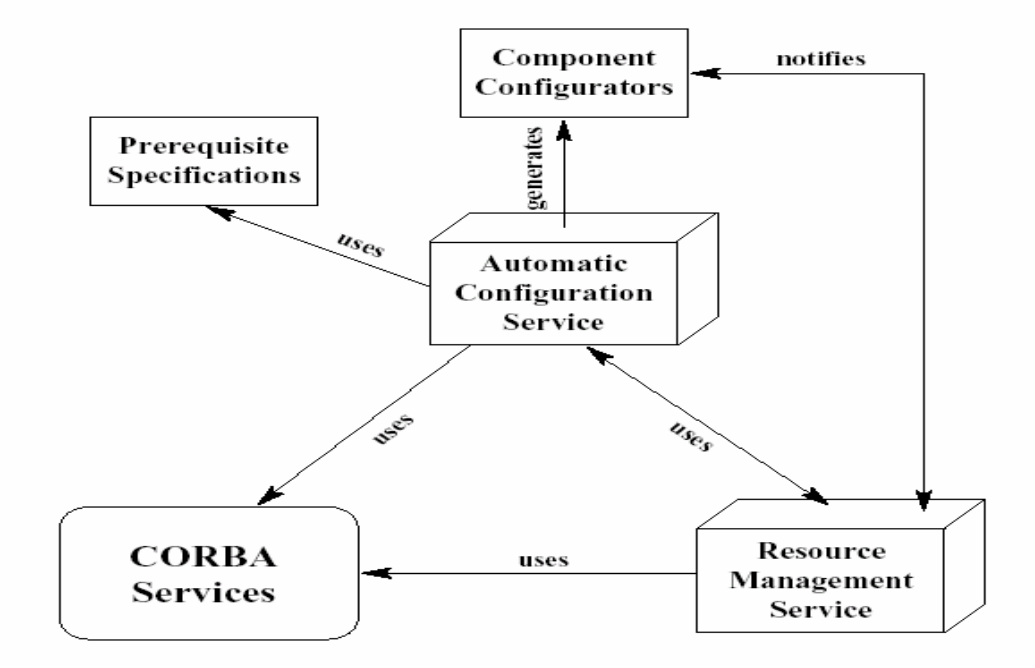

Figure 2.3. Architectural Framework of the 2K Operating System

Thus, the above-illustrated services form the underlying mechanism to realize the QoS objectives of the 2K operating system.

2.2.3. Nemesis Operating System

[LES96] describes an entirely new operating system called Nemesis whose design is geared to the support of time-sensitive applications requiring a consistent QoS, such as those that use multimedia. It is intended to provide guaranteed fine-grained levels of system resources like CPU, memory and disk bandwidth.

The principle behind Nemesis is to design the operating system in a way that would allow a majority of the application code to execute in the application process itself, instead of the kernel. This has led to a small lightweight kernel, with most operating system functions being performed in shared libraries that execute in the user's process. Due to this feature, Nemesis has been classified as a vertically structured operating system.

The ability to provide guaranteed QoS often comes with the penalty of overhead due to frequent context-switches. Nemesis addresses this problem by using a single address space, which leads to reduced memory-related context-switch overhead.
The QoS management in Nemesis is performed using feedback control. This approach is adaptive and involves a controller adjusting application QoS demands according to the measured performance. Figure 2.4 indicates the structure of the QoS feedback control mechanism.

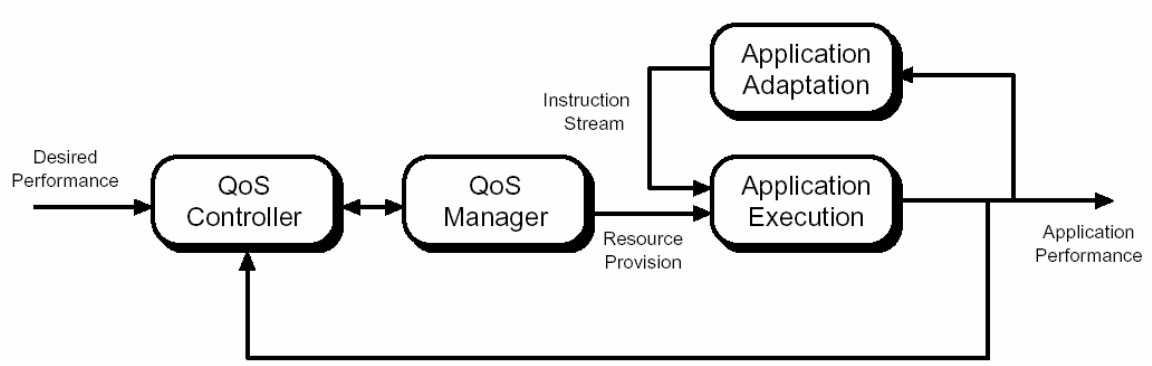

Figure 2.4. QoS Feedback Control

Here, the QoS Controller dictates the QoS policy to be followed and it can be directly controlled by the user or by an agent running on the user's behalf. The QoS Manager is responsible for implementing the allocation of resources to achieve the QoS policies supplied by the QoS controller. The QoS Manager ensures that these policies are enforced by informing the operating system and the applications to suitably adapt their behavior.

By using this approach, the application developers are freed from the task of determining exactly what resources an application requires. It also simplifies the task of porting the applications to new environments. However, it requires implementing adaptive algorithms and defining the QoS policies.

This section describes a few Operating systems with QoS guarantees. Most of the operating systems that are QoS enabled are part of a distributed system as opposed to a non-distributed system. Hence, these operating systems often have to rely on the

underlying network for communication and utilize some form of remote communication mechanism like RPC (Remote Procedure Call) or RMI (Remote Method Invocation). This means that these operating systems often rely on the QoS mechanisms of the underlying network, dealt with in section 2.1, in order to achieve their QoS guarantees. Subsequently, all the layers above the operating system, either directly or indirectly, rely on operating systems that support QoS enforcement in order to realize their QoS objectives at the end systems.

2.3. QoS in Middleware

Several efforts have been made to provide QoS provisions in middleware which resides between applications and operating system kernels. The most significant among these are outlined below.

2.3.1. The ACE ORB (TAO)

TAO was the first ORB to support end-to-end QoS guarantees over ATM/IP networks. "TAO is an open-source standards-based, high-performance, real-time ORB end-system communication middleware that supports applications with deterministic and statistical QoS requirements, as well as 'best-effort' requirements" [SCH98]. It was developed using the ACE framework [ACE02], which is described as a highly portable middleware communication framework. ACE contains a set of C++ components that are used to realize strategic design patterns for high performance and real-time communication systems.

TAO enhances the standard CORBA Event Service to provide important features, such as real-time event dispatching and scheduling, periodic event processing, efficient event filtering and correlation mechanisms, and multicast protocols required by real-time applications. According to [SCH98], the main objectives of TAO are as follows:

1. Identification of enhancements to OMG CORBA specifications that would enable applications to precisely state their QoS requirements to ORB end systems.
2. Empirical determination of the features required for real-time ORB endsystems that can enforce application QoS guarantees.
3. Integration of ORB middleware with the I/O subsystem architectures and optimization strategies to provide guaranteed end-to-end bandwidth, latency and reliability.
4. To capture and document the key design patterns necessary to develop, maintain, configure and extend real-time ORB endsystems.

TAO's ORB endsystem contains the following sub-systems:

1. I/O Subsystem: It is responsible for sending and receiving requests to and from clients, in real-time, across a network.
2. Run-time Scheduler: It is used to determine the priority at which requests are processed by clients and servers in an ORB endsystem.
3. ORB Core: It is a flexible, portable and predictable CORBA inter-ORB protocol engine that delivers client requests to the Object Adapter described below and returns responses to the clients.
4. Object Adapter: It is used to demultiplex and dispatch client requests to servers using hashing.
5. Stubs and skeletons: These are used to optimize the primary sources of marshaling and demarshaling overhead in the code automatically generated by TAO's IDL compiler.
6. Memory Manager: It minimizes the sources of dynamic memory allocation and data copying throughout the ORB end-system.
7. QoS API: It allows applications and higher-level CORBA services to specify their QoS parameters using an object-oriented approach.

TAO uses an extension of CORBA Interface Definition Language (IDL) called Real-time Interface Definition Language (RIDL) to represent QoS requirements of

applications. The IDL extensions RT_Operation Interface and RT_Info struct are used to convey QoS information like CPU requirements to the ORB on a per-operation basis.

The RT_Operation interface is the mechanism for conveying CPU requirements of applications to TAO's scheduling service. It contains type definitions of the entities used in the QoS mechanism. A sample CORBA IDL description of the RT_Operation Interface schema as represented in [SCH98] is shown in figure 2.5.

```
module RT_Scheduler
{
// Module TimeBase defines the
//OMG Time Service.
typedef TimeBase::TimeT Time;
//100 nanoseconds
typedef Time Quantum;
typedef long Period;
// 100 nanoseconds
enum Importance
// Defines the importance of the
//operation,
// which can be used by the
//Scheduler as a
// "tie-breaker" when other
//scheduling
// parameters are equal.
{
VERY_LOW_IMPORTANCE,
LOW_IMPORTANCE,
MEDIUM_IMPORTANCE,
HIGH_IMPORTANCE,
VERY_HIGH_IMPORTANCE
};
typedef long handle_t;
// RT_Info's are assigned per-
//application unique identifiers.
struct Dependency_Info
{
long number_of_calls;
handle_t rt_info;
// Notice the reference to the
//RT_Info we depend on.
};
typedef sequence<Dependency_Info>
Dependency_Set;
typedef long OS_Priority;
typedef long Sub_Priority;
typedef long Preemption_Priority;
struct RT_Info
// = TITLE
// Describes the QoS for an
//"RT_Operation".
// = DESCRIPTION
// The CPU requirements and QoS
//for each "entity" implementing
//an application
// operation is described by the
//following information.
{// Application-defined string that
//uniquely identifies the
//operation.
string entry_point_;
// The scheduler-defined unique
//identifier.
handle_t handle_;
// Execution times.
Time worstcase_execution_time_;
Time typical_execution_time_;
// To account for server data
//caching.
Time cached_execution_time_;
// For rate-base operations, this
//expresses the rate. 0 means
//"completely passive",
// i.e., this operation only
//executes when called.
Period period_;
// Operation importance, used to
//"break ties".
Importance importance_;
// For time-slicing (for BACKGROUND
// operations only).
Quantum quantum_;
// The number of internal threads
//contained by the operation.
long threads_;
// The following parameters are
//defined by the Scheduler once the
//off-line schedule is computed.
// The operations we depend upon.
Dependency_Set dependencies_;
// The OS priority for processing
//the events generated from this
//RT_Info.
OS_Priority priority_;
// For ordering RT_Info's with
//equal priority.
Sub_Priority subpriority_;
// The queue number for this
//RT_Info.
Preemption_Priority
preemption_priority_;
};
};
```

Figure 2.5. RT_Operation Interface Schema in TAO

The applications that use TAO need to specify their resource requirements before execution. The RT_Info IDL struct is used to express these requirements. The following parameters are used to illustrate a RT_Info struct in case of CPU scheduling:

Worst-case Execution Time: It is the maximum execution time that the RT_Operation requires. It is used for conservative scheduling analysis in real-time applications.

Typical Execution Time: It is the time taken normally for execution of RT_Operation.

Cached Execution Time: It is set to a non-zero value depending on whether an operation can provide a cached result in response to requests. For operations that are periodic, the worst-case execution cost is incurred only once per period if this field is non-zero.

Period: It is defined as the minimum time between successive iterations of an operation.

Criticality: Criticality of an operation is an enumeration value ranging from the lowest possible criticality (VERY_LOW_CRITICALITY), to the highest possible criticality (VERY_HIGH_CRITICALITY). Criticality is used as the primary consideration while assigning priority to operations.

Importance: Operation importance is another enumeration parameter with values ranging from lowest importance (VERY_LOW_IMPORTANCE), to the highest importance (VERY_HIGH_IMPORTANCE). The importance of an operation is used as a tie-breaker to order the execution of RT_Operations when criticality fails to resolve operation priority.

Quantum: It is defined as the maximum time that an operation is allowed to run before preemption, in case there are other operations with the same priority. This time-sliced scheduling is used to prevent starvation of low priority operations.

Dependency Info: It is a set of handles to other RT_Info instances, one for each RT-Operation that it depends on. The Dependency Info is used during scheduling to identify threads within the system, with each dependency graph representing a thread.

The RIDL schemas RT_Operation and RT_Info are used to specify the run-time execution characteristics of object operations to TAO's scheduling service. TAO uses this information to validate the feasibility of a schedule and allocate ORB endsystem and network resources to provide the desired QoS.

2.3.2. Quality of Service Management Environment (QoSME):

QoSME is intended to serve the end application QoS by providing an architecture open to applications and network systems [WAN00]. It originated from the QUAL (Quality of Service Assurance Language) project [FLO96].

QoSME provides the QoS to applications by requesting resource allocations from underlying service providers such as IntServ, DiffServ and ATM. It provides different modes of service Guaranteed Service, Controlled Load Service which are (inherited from IntServ/ RSVP) and Hard and Soft modes. Guaranteed Service and Controlled Load Service are described in section 2.1.2. The Hard mode is used for QoS provisioning in the ATM switch network and the Soft mode is used for QoS provisioning by IntServ/RSVP, on the Internet.

QoSME maps the application QoS requirements to QoS parameters of RSVP or ATM. If a resource reservation is possible, it associates the network connections of that application with this reservation.

A QoSME application defines its QoS requirements using QoSME APIs or QUAL and obtains QoS guarantees via QoSockets which are a modified version of Berkeley sockets with support for QoS. QoSockets compile the application QoS specifications into respective transport protocols and mechanisms and also provide the instrumentation to monitor the QoS delivered to the application. A sample QUAL specification of network level QoS measures as depicted in [FLO96] is shown in figure 2.6.

```
% Receiver Process
realtm { loss 6;          /* The percentage of messages lost during transmission
                          must not be higher than 10^-6 */
     permt;               /* Permutation is allowed in the transmission, i.e., messages
                          need not to be delivered in the order they were sent */
     rate  sec 10 -
           sec 15;        /* Mean transmission rate is between 10 and 15 messages/sec */
     peak - sec 20;       /* Peak transmission rate is 20 messages/sec */
     delay ms 35;         /* Transmission delay must be less than 35ms */
     inter-delay ms 33;   /* Inter-message delay must be less than 33ms */
     recovery sec 3;}     /* Any recovery must take less than 3sec */
  handlers { net_handler
        {manage_conn;};}  /* Port that receives network level QoS violations */
  receiveport              /* Keyword that identifies a port declaration */
    {video_frame_t}        /* Type of messages exchanged through the port */
    video_input;           /* Port identifier */

% Sender Process
realtm { loss NULL;
     permt NULL;          /* No constraints regarding loss or permutation */
     rate - sec 25;       /* Message mean transmission rate is 25 messages/sec */
     peak - sec 30;       /* Peak rate is 30 messages/sec */
                          /* No constraints regarding delay or inter-message delay */
     recovery sec 4;}     /* Any recovery must take less than 4sec */
  receiveport {video_frame_t}
    *video output;        /* The symbol * distinguishes an outport from an inport */
```

Figure 2.6. Network level QoS specification in QUAL

The main QoS parameters supported by QoSME are:

- Throughput: The four parameters used to represent throughput are as follows:
    - Min_rate, which is the lower bound on transmission rate.
    - Max_rate, which is the upper bound on transmission rate.
    - Peak_rate, which is the peak transmission rate.
    - Size, which is the maximum size of transmitted messages.

Throughput is calculated as the product of the rate (messages per second) and the message size (in bytes).

- Delay and jitter: The parameters related to delay and jitter are:
    - Min_delay, which is the lower bound on transmission delay.
    - Max_delay, which is the upperbound on transmission delay.
    - Int_delay, which is the maximum time delay variance of two consecutive messages.
- Reliability: The parameters utilized for reliability are:
    - Loss, which is the percentage of messages that are lost.
    - Rec_time, which is the maximum time elapsed for recovering a disrupted connection.
    - Permt, which is a permutable flag indicating if messages can be delivered out of order.
- Coerced flags: QoSME allows both senders and receivers of a stream to define their own parameters. Hence, there is a possibility that QoS parameters at each end conflict. This situation is handled by coercing or downgrading the conflicting parameters to a commonly accepted level. The coerced flags are used to indicate which parameters are to be coerced.

QoSME, thus provides the support to map the application QoS requirements stated in terms of the above mentioned QoS parameters, using the QUAL, into the QoS parameters of RSVP or ATM.

2.3.3. RAPIDware

The RAPIDware project is intended to address the design and implementation of adaptive, component-based middleware services for dynamic, heterogeneous environments [MCK01]. The main goal of the RAPIDware project has been to develop adaptive mechanisms and programming abstractions that enable middleware frameworks to execute in an autonomous manner, by dynamic instantiation and reconfiguration of components in response to changing client demands. The RAPIDware project finds its roots in the Pavilion framework [MCK99] which is an object-oriented framework supporting synchronous web-based collaboration. The Pavilion framework has been extended by RAPIDware by introduction of programming abstractions and mechanisms to automate the instantiation and reconfiguration of middleware components in order to accommodate hosts with limited resources. The configuration of RAPIDware middleware adaptive components as depicted in [MCK01] is shown in figure 2.7 (reproduced from [MCK01] with permission).

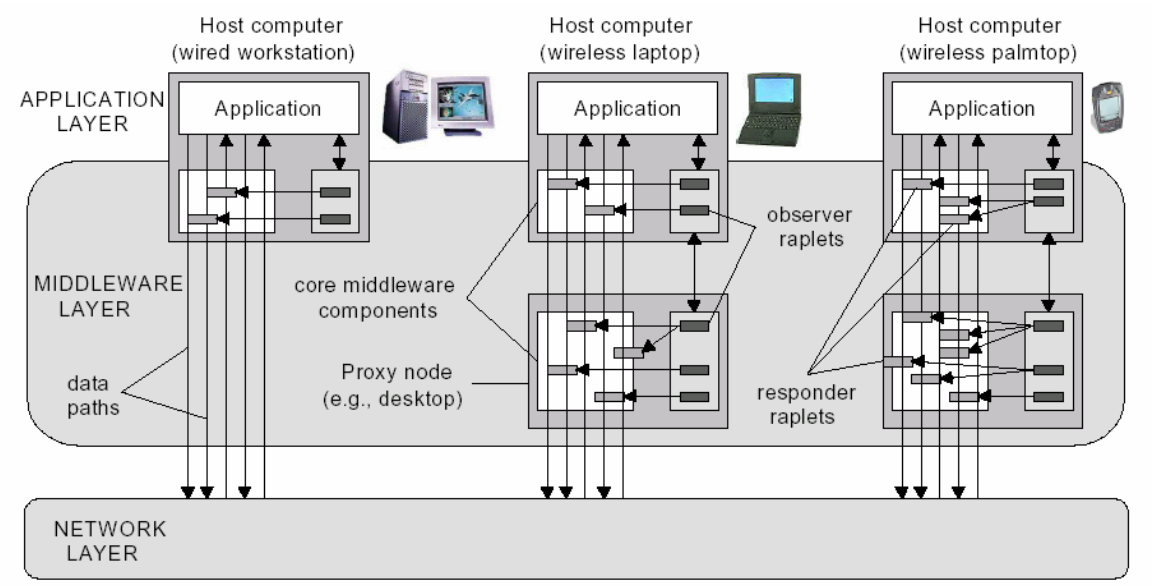

Figure 2.7. Configuration of RAPIDware adaptive middleware components

The separation of adaptive components from non-adaptive components is a key principle in this approach. Here, the adaptive components are referred to as *raplets*. Two

categories of raplets are used namely, *observers* and *responders*. The observers are responsible for collectively monitoring the system state. On detection of a relevant event, the observer will either instantiate a new responder or request an existing responder to address the event by taking the necessary action. The responders in turn handle these events by instantiating new components or by modification of communication protocol behavior.

The collection of interfaces provided by a set of meta-objects is called meta-object protocol (MOP) [KAS02]. RAPIDware proposes a model for adaptive components that is designed to facilitate the construction and evolution of MOPs for QoS, fault tolerance, and security. The concept of providing separate component interfaces for observing behavior (introspection) and for changing behavior (intercession). Here, a component contains two types of primitive operations: refractions, which provide a glimpse of the underlying base-level component and transmutations, which modify the functionality of the base-level component. [KAS02] describes an extension to Java language called Adaptive Java as a prototype to identify the language constructs that are necessary in dynamic and adaptive languages. An Adaptive Java component structure is similar to that of a Java class with the standard Java methods being replaced by invocations and standard immutable variable declarations supplemented with mutable variable declarations. The structure of an Adaptive Java component as depicted in [KAS02] is shown in figure 2.8.

```
/* A simple component */
component BasicComponent {
  /* Constructor */
  public BasicComponent() { ... }

  /* Invocation  */
  public invocation void
       method1(String arg) { ... }
            .
            .
            .
}
```

Figure 2.8. Adaptive Java Component Structure

[KAS02] also proposes an extension to regular socket classes called *metasockets* using Adaptive Java. An application can modify the metasocket functionality y using refractions and transmutations. The structure of a metasocket for a wireless audio streaming application as depicted in [KAS02] is shown in figure 2.9 (reproduced from [KAS02] with permission).

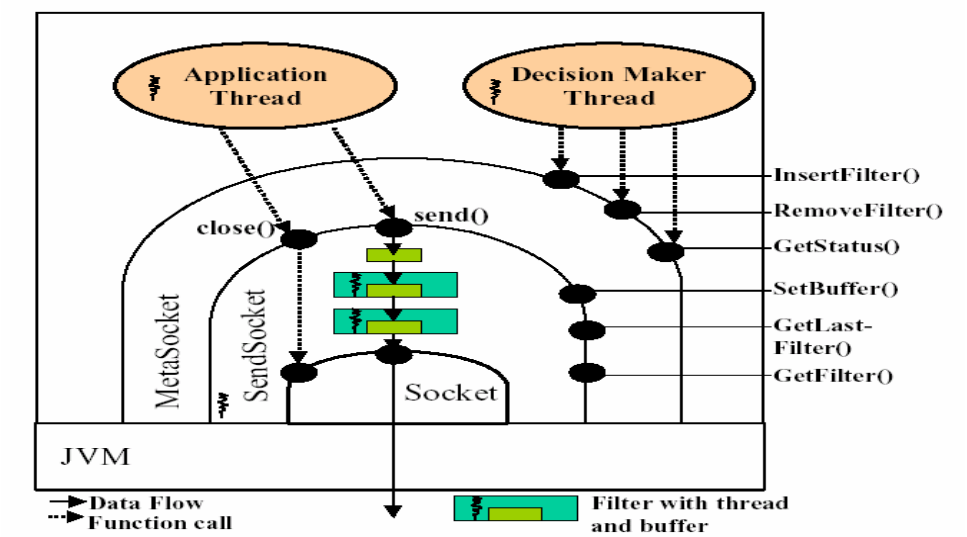

Figure 2.9. Structure of a metasocket

The base component called *sendSocket* is the Java Socket class. *Send( )* and *close( )* are the invocations available to external components. *SendBuffer*, *GetFilter*, *GetLastFilter* are intended for use by the meta level. *GetStatus( )* is refraction used to obtain the current configuration of filters. *InsertFilter( )* and *RemoveFilter( )* are transmutations used to modify the filter pipeline.

Adaptive Java and the metasockets together provide the low-level mechanisms necessary for construction of meta-object protocols for concerns like quality of service, security and fault tolerance.

2.3.4. OpenORB

OpenORB is a middleware platform incorporating reflection using a component-based approach [BLA01]. The key principle behind reflection is to provide a meta-interface supporting the inspection and adaptation of the underlying virtual machine. The

meta-interface is intended to support operations that allow discovery of the internal operation and structure of the middleware platform, like the deployed protocols and management structures and allow changes to be made to the system at runtime.

In OpenORB, every application-level component has a meta-interface through which it can access the underlying metaspace that provides the support environment for the component. The Metaspace itself is, in turn, composed of meta components which allow access to their support environment through meta-interfaces. This results in recursive levels of reflection. In practice, the metacomponents are instantiated on demand so, unless accessed, they only exist in theory. For the sake of separation of concerns between different system aspects, the metaspace is divided into two metaspace models namely: metaspace models for structural reflection and metaspace models for behavioral reflection.

Structural reflection refers to the content and structure of a component. In OpenORB the structural reflection metaspace is represented by two metamodels, the interface and architecture metamodels. The interface metamodel provides the external view of a component and allows access to the set of interfaces of the component. The architecture metamodel provides a view of the internal structure of a component and allows access to the implementation of the component.

Behavioral reflection refers to the activities within the underlying system. In OpenORB the behavioral reflection metaspace is represented by the metamodels, namely, interception and resources. Interception metamodel is used to introduce monitoring and accounting into a system by using pre and post conditions. The resources metamodel is used for management of the resources required to complete the activities specified by the interception metamodel.

OpenORB uses an architectural description language (ADL) called Xelha [CAZ99]. The main objective of Xelha is to support the management of resource concerns in distributed real-time systems. In OpenORB, the ADL is utilized to specify QoS requirements. These QoS requirements are then used to obtain the corresponding resource requirements for a task. ADL also has the facilities to support dynamic

monitoring and controlling of components. A sample use of Xelha as depicted in [BLA01] is shown in figure 2.10:

```
Def connector <stream> AudioConnector_V1(string srcCapsule, string sinkCapsule):
  components:
      srcStub: SrcStub, srcCapsule
      sinkStub: SinkStub, sinkCapsule
  connectors:
      streamConn: StreamConnector(srcCapsule, sinkCapsule)
  interfaces:
      interaction:
                IN: SrcStubIN, ( srcStub, IN )
            OUT: SinkStubOUT, ( sinkStub, OUT )
      control:
            CTRL: StreamConnCTRL, (streamConn, CTRL)
  composition graph:
     interfaces:
            OUT: ( srcStub, OUT )
           streamConnIN: ( streamConn, IN )
          streamConnOUT: ( streamConn, OUT )
        IN: ( sinkStub, IN )
     edges:
        ( OUT, streamConnIN )
              ( streamConnOUT, IN )
  tasks:
       Def task transmitAu.marshall:
              switching points:
    srcStub:CTRL:start [if taskx]
  qos specifications:
             delay(srcStub:IN:read, streamConn:IN:put) = 5
                  throughput(srcStub:OUT:put) = 64
 Def task transmitAu includes transmitAu.marshall, transmitAu.unmarshall:
     importance: 5
     qos specifications:
                  delay(streamConn:IN:put, streamConn:OUT:put) = 10
                 packet_loss(streamConn:IN:put, streamConn:OUT:put) = 5
                delay(srcStub:IN:read, sinkStub:OUT:write) = 20
                 jitter(srcStub:IN:read, sinkStub:OUT:write) = 1
                qos management structure: ...
                 <not shown for simplicity>
```

Figure 2.10. Sample use of Xelha

The QoS requirements in OpenORB are specified using Xelha as shown above. These QoS requirements are then used to derive the underlying resource allocation

policies for tasks. This ensures that the tasks are allocated sufficient resources to meet the QoS requirements.

In this section, a few of the significant approaches to QoS-enabled middleware were presented. Middleware plays a significant role in a DCS by providing the application developer with convenient application programming interfaces (APIs) (like sockets) to build distributed applications. In order for distributed applications to deliver guaranteed QoS, the middleware that the application uses must also support QoS. Hence, QoS enabled middleware is essential to building distributed applications with QoS guarantees.

2.4. QoS in Applications (End-to-End QoS)

This section deals with QoS efforts focused on building distributed applications with QoS guarantees. This involves the work related to assurance of end-to-end QoS, i.e., the QoS delivered to the end-user.

2.4.1. Quality Objects (QuO)

The Quality objects (QuO) framework [BBN01] provides the QoS to distributed software applications composed of objects. QuO is intended to bridge the gap between the socket-level QoS and the distributed object level QoS. This work mainly emphasizes the specification, measurement, control and adaptation to changes in quality of service.

QuO extends the CORBA functional IDL with a QoS description language (QDL). QDL is a suite of quality description languages for describing QoS contracts between clients and objects, the system resources and mechanisms for measuring and providing QoS and adaptive behavior on the client and object side. It utilizes the Aspect-Oriented Programming paradigm, which provides support for incorporating the non-functional properties of components separately from the functional properties.

QDL consists of a set of two languages, a Contract Description Language (CDL) and a Structure Description Language (SDL) [LOY98]. CDL is used to specify a QoS

contract between a client and object in an application. This contract describes the QoS desired by the client and the actual QoS the object expects to provide. The contract is expressed in terms of a set of operating regions, the behavior to be invoked in order to adapt to changes in QoS and to notify the interfaces to the elements of the system that can be used to measure and control QoS. A CDL contract consists of the following elements:

a) A set of nested QoS states represented by operating regions with each operating region being assigned a predicate, indicating whether it is active or not.
b) Target behaviors to trigger in case of changes in states of operating regions.
c) References to system condition objects passed as parameters to the contract or declared locally in the contract. The system condition objects are used to obtain values of system resources, client state etc.
d) Callback objects passed in as parameters to the SDL and used to notify the clients about state transitions.

A sample CDL contract as illustrated in [LOY98] is shown in figure 2.11:

```
contract Replication(
  syscond ValueSC ValueSCImpl ClientExpectedReplicas,
  callback AvailCB ClientCallback,
  syscond ValueSC ValueSCImpl MeasuredNumberReplicas,
  syscond ReplSC ReplSCImpl ReplMgr ) is

  negotiated regions are
    region Low_Cost : when ClientExpectedReplicas == 1 =>
      reality regions are
        region Low : when MeasuredNumberReplicas < 1 =>
        region Normal : when MeasuredNumberReplicas == 1 =>
        region High : when MeasuredNumberReplicas > 1 =>
        transitions are
          transition any->Low : ClientCallback.availability_degraded();
          transition any->Normal : ClientCallback.availability_back_to_normal();
          transition any->High : ClientCallback.resources_being_wasted();
        end transitions;
      end reality regions;
    region Available : when ClientExpectedReplicas >= 2 =>
      reality regions are
        region Low : when MeasuredNumberReplicas < ClientExpectedReplicas =>
        region Normal : when MeasuredNumberReplicas >= ClientExpectedReplicas =>
        transitions are
          transition any->Low : ClientCallback.availability_degraded();
          transition any->Normal : ClientCallback.availability_back_to_normal();
        end transitions;
      end reality regions;
    transitions are
      transition Low_Cost->Available :
        ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
      transition Available->Low_Cost :
        ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
    end transitions;
  end negotiated regions;
end repl_contract;
```

Figure 2.11. Sample CDL contract

The above contract specifies the replication behavior of a QuO application. Here, the client has two different operating modes corresponding to low and high levels of availability. The client can request either one replica (represented by the Low_Cost region) or multiple replicas (represented as the Available region) depending on its requirements.

SDL allows the specification of adaptation alternatives and strategies depending on the measured QoS of the system. An SDL description consists of the following elements:

a) The set of interfaces and contracts whose adaptive behavior is being specified by the SDL specification.
b) A list of method calls and returns for which the adaptive behavior is to be specified.
c) A list of regions representing the states of QoS that could be triggered by adaptive behavior.
d) A set of behavior specifications which can specify choosing between alternate object bindings , creating new bindings, choosing between alternate methods, throwing an exception or executing a piece of code.
e) A set of default behavior specifications to used for those method calls or contract regions not explicitly listed.

[LOY98] illustrates a sample SDL that chooses between replicated and non-replicated server objects as shown in figure 2.12.

```
delegate behavior for Targeting and Replication is
  obj : bind Targeting with name SingleTargetingObject;
  group : bind Targeting with characteristics { Replicated = True };

  call calculate_distance_to_target :
    region Available.Normal :
      pass to group;
    region Low_Cost.Normal :
      pass to obj;
    region Available.Low :
      throw AvailabilityDegraded;
    default :
      pass to obj;
  return calculate_distance_to_target :
    pass through;
  default :
    pass to obj;
end delegate behavior;
```

Figure 2.12. Sample SDL specification

This framework is centered on the notion of a connection between a client and an object. Here, a connection is primarily a communication channel with QoS awareness. QoS regions are predicates of measurable connection properties such as throughput and

jitter. The level of QoS is continuously monitored once the connection is established. If the measured QoS is found to be outside the expected region, the client is informed through an upcall. The client and object now try to adapt to the new conditions and renegotiate a new expected region.

2.4.2. Quality of Service Modeling Language (QML):

QoS Modeling Language (QML) is a QoS specification Language proposed in [FRO98]. QML is an extension of UML. It is a general purpose QoS specification language capable of describing different QoS parameters in any application domain.

QML offers three main abstraction mechanisms for QoS specification: contract type, contract and profile. A contract type represents a specific QoS category like reliability or performance, and it defines dimensions that can be used to characterize a particular QoS category. Dimensions are factors that determine a given QoS category. For example, delay and throughput are considered to be dimensions of the QoS category performance. A contract is defined as an instance of a contract type and it represents a particular QoS specification. A contract generally contains a set of constraints imposed upon the values of the dimensions of a QoS category. Profiles are used to associate contracts with interface entities such as operations, operation arguments and operation results. Profiles are generally defined for specific interfaces and specify the QoS contracts for the categories and operations described in the interface. An interface can have multiple profiles depending on the number of implementations of the interface. A profile can be used either to specify client QoS requirements or to specify QoS provisioning. [FRO98] illustrates contract types, contracts and profiles using the following example:

```
type Reliability = contract {
        numberOfFailures: decreasing numeric no/year;
        TTR: decreasing numeric sec;
        availability: increasing numeric;
};
```

```
type Performance = contract {
        delay: decreasing numeric msec;
        throughput: increasing numeric mb/sec;
};

systemReliability = Reliability contract {
        numberOfFailures < 10 no/year;
        TTR {
                percentile 100 < 2000;
                mean < 500;
                variance < 0.3
              };
        availability > 0.8;
};

rateServerProfile for RateServiceI = profile {
        require systemReliability;
        from latest require Performance contract f
                delay {
                        percentile 50 < 10 msec;
                        percentile 80 < 20 msec;
                        percentile 100 < 40 msec;
                        mean < 15 msec
                      };
};
from analysis require Performance contract {
        delay < 4000 msec
        };
};
```

Figure 2.13. A sample QML description

The figure 2.13 defines the QoS requirements of the Rate Service subsystem of a Currency trading system. The Rate Service system provides the rates, interests and other information important to foreign exchange trading.

A CORBA IDL interface for Rate Service is shown in figure 2.14.

```
interface RateServiceI {
        Rates latest (in Currency c1,in Currency c2) raises (InvalidC);
        Forecast analysis (in Currency c) raises (Failed);
};
```

Figure 2.14. CORBA IDL interface for Rate Service

Here, Reliability and Performance are two QoS categories under consideration, as illustrated by the *Reliability* and *Performance* contract types. *Reliability* depends on the dimensions *numberOfFailures*, *TTR* (Time to Recovery) and *availability*, as illustrated in the *Reliability* contract type. Also, the *Performance* depends on the dimensions *delay* and *throughput* as illustrated in the *Performance* contract type.

SystemReliability is an instance of the *Reliability* contract type and describes the specific limits imposed upon the values of the dimensions of *Reliability*.

rateServerProfile defines the QoS requirements expected from the individual operations of the *RateServiceI* interface, namely, *latest* and *analysis*.

In QML, the QoS specifications are syntactically separate from interface definitions, allowing different implementations of the same service interface to have different QoS characteristics. Thus, a service specification may consist of a functional interface and one or more QoS specifications.

QML provides a means to express the QoS requirements of distributed object systems. However, it assumes that the mechanisms for the monitoring and the adaptation of QoS

are provided by the underlying middleware and does not provide any mechanisms to address these issues.

2.4.3. Quality of Service Architecture (QoS-A):

[CAM96] proposes a quality of service architecture (QoS-A) to specify and achieve the necessary performance properties of continuous media applications over asynchronous transfer mode (ATM) networks. In QoS-A, instead of considering the QoS in the end-system and the network separately, a new integrated approach, which incorporates QoS interfaces, control, and management mechanisms across all architectural layers, is used. This architecture is based on the notions of the flow, the service contract and the flow management.

Flows characterize the production, transmission and consumption of single media streams with associated QoS. A service contract makes it possible to formalize the QoS requirements of the user and the potential degree of service commitment of the service provider. The service contracts are predefined C-language structs that allow the specification of QoS parameters like throughput, delay, jitter and loss. It also enables the specification of the network resource requirements and the necessary remedial actions to be taken in case of a service contract violation. The remedial actions may involve adjusting internal state to accommodate current load conditions, renegotiating the flow QoS, dropping components of a multi-layer coded flow - for example dropping MPEG enhancements or disconnecting from service. The flow management is utilized to monitor and maintain the QoS specified in the service contract.

The QoS-A is composed of a number of layers and planes. The upper layer consists of a distributed application platform along with the services that provide multimedia communications. The orchestration layer is present below the platform layer and it provides jitter correction and multimedia synchronization services across multiple related flows. Below this is the transport layer which contains a range of QoS configurable services and mechanisms. The network layer, the data link layer and the

physical layer appear in that order below the transport layer. QoS-A incorporates QoS management in three vertical planes, namely,

- the protocol plane which consists of distinct user and control sub-planes
- the QoS maintenance plane and
- the flow management plane

The user plane allows the user to select upcalls for notification of corrupt and lost data at the receiver and also allows negotiation of QoS parameters like bandwidth, jitter and delay. The control plane is responsible for establishment of point-to-point and multicast connections and signaling support for dynamic QoS management required by the flow management plane. The QoS maintenance plane consists of a number of QoS managers that are responsible for the fine grained monitoring and maintenance of their associated protocol entities (the QoS-A layers). For instance, at the orchestration layer, the QoS manager is concerned with the tightness of synchronization between multiple related flows. While, at the transport layer, the QoS manager handles the intra-flow bandwidth, loss, jitter and delay. The QoS Managers maintain the level of QoS by means of fine grained resource tuning strategies based on flow monitoring information. The flow management plane is responsible for flow establishment which includes QoS-based routing and resource reservation, and QoS mapping between layers.

In the QoS-A, a Service Contract is used to formalize the QoS requirements of the user and the potential degree of service commitment of the service provider. In implementation, the service contract is expressed as a C language struct. A sample service contract as illustrated in [CAM96] is shown in figure 2.15.

```
typedef struct {
flow_spec_t flow_spec;
commitment_t commitment;
adaptation_t adaptation;
maintenance_t maintenance;
cost_t cost;
} service_contract_t;
```

Figure 2.15. Sample service contract in QoS-A

In the above service contract,

- flow_spec_t : specifies the user's traffic performance requirements.
- commitment_t: indicates the degree of resource commitment required from the lower layers.
- adaptation_t: specifies the actions to be taken in case of violations of service contract.
- Maintenance_t: indicates the required degree of monitoring and maintenance.
- Cost_t: indicates the costs the user is willing to pay for the services requested.

Thus, using the concepts of service contracts, flow and flow management, QoS-A provides a framework to incorporate QoS guarantees into continuous media applications.

2.4.4. ISO/IEC 9126

[ISO99] lists a set of quality characteristics for software products. It is an attempt to define QoS parameters and provide some simple measurement rules for evaluating the software quality. It categorizes software quality into six characteristics namely: functionality, reliability, usability, efficiency, maintainability and portability. These characteristics are further divided into other sub-characteristics. The QoS Catalog which is a part of the UQOS framework shares some of the same objectives as [ISO99]. The [ISO99] includes a higher number of quality characteristics (if including the sub-characteristics) than the QoS Catalog. However, the QoS Catalog provides much more information about each of the QoS parameters. As stated in section 4.1., one of the objectives of the QoS Catalog is to act as a comprehensive source of information about each of the QoS parameters. The QoS Catalog tries to achieve this objective, by providing a lot more information about each QoS parameter than provided in [ISO99]. Also, as stated in section 4.1., the inclusion of the parameters into the QoS Catalog is considered to be an evolving process. In order to further validate this claim a feature-wise comparison of the QoS Catalog and the [ISO99] is provided in table 2.1:

Table 2.1. Comparison of the features of the QoS Catalog and the ISO/IEC9126

Feature	QoS Catalog	ISO/IEC9126
Intent	Yes	Yes
Description	Yes	Yes
Motivation	Yes	No
Applicability	Yes	No
Model used	Yes	No
Influencing factors	Yes	No
Measuring unit	Yes	No
Evaluation Procedure	Yes	Yes
Evaluation Formulae	Yes	Yes
Result Type	Yes	Yes
Static/Dynamic	Yes	No
Increasing/Decreasing	Yes	Yes
Composable/Non-composable	Yes	No
Consequences	Yes	No
Related Parameters	Yes	No
Domain of usage	Yes	No
User Caution	Yes	No
Aliases	Yes	No

Further, the [ISO99] is intended for software in general and does not address the issues unique to CBSD. Some of these issues specific to CBSD which are addressed by the UQOS are: the issue of composition and decomposition of QoS parameters, the issue of the effect of environment on the QoS, the issue of effect of usage patterns on the QoS and the issue of specification of the QoS of software components. These issues are explained in more detail in chapter 4.

Thus the UQOS framework provides a QoS Catalog that is more comprehensive than the [ISO99] and in addition, it also addresses the issues unique to QoS in CBSD.

In this section, a few important efforts related to QoS in applications were discussed. Realizing the QoS at the application level involves ensuring that all the other layers lower down in the hierarchy also support the notion of QoS. Hence, in order to fully exploit application level QoS (end-to-end QoS), all the layers of the DCS including the application (dealt with in this section), middleware (dealt with in section 2.3), the operating system (dealt with in section 2.2) and the underlying network (dealt with in section 2.1) must be QoS enabled.

2.5. QoS in Software Components

In the previous section the efforts related to QoS in applications were discussed. Component-Based Software Development (CBSD) is now seen as the future trend in the world of software. In order for this approach to result in reliable software, the software components utilized should, in turn, offer a guaranteed level of quality. This requires a comprehensive QoS framework for software components. However, very few of the existing technologies offer a QoS framework directed towards software components and the unique issues and challenges arising out of CBSD, such as:

i. Notion of quality: In the realm of CBSD, the system developer (who assembles the end system from individual components) uses components created by various component developers. However, there is no consensus among the component developers as to what exactly constitutes the “quality” of a software component.

ii. QoS Quantification: The quantification of QoS is a quintessential part of any QoS framework. However, in the world of CBSD, with a plethora of component developers, the quantification of component quality is carried out in an adhoc manner, if at all. Hence, there is a lack of standardization of QoS quantification schemes.

iii. Effect of environment: According to the CBSD philosophy, a given software component may be used in diverse environments (CPU, memory, system bus, operating system, priority schemes, etc). However, this raises the question of how the environment might affect the QoS of the component. This is especially true for dynamic QoS parameters (parameters whose values depend on environmental conditions).

iv. Effect of Usage patterns: A component developer often has no control over how the component may be used once it is deployed. This means that the component is often exposed to varying usage patterns depending on its deployment site, the semantics of the application and the time of the day. This necessitates the study of the effect of Usage patterns on the QoS of a component.

v. Effect of Composition/Decomposition: In CBSD, several components with varying QoS levels are composed together to build an end-system. This raises the question of determining the effect of composition on the QoS of the end-system, i.e., obtaining the QoS of the end-system, given the QoS of individual components. Similarly, it is necessary to be able to deduce the QoS of the constituent components, given the QoS of the end-system. This is essential to produce the QoS-based search criteria for the headhunters.

The UniFrame approach to QoS (introduced in chapter 1) tries to address these issues in the following manner:

i. By creation of a QoS Catalog for software components containing detailed descriptions of QoS parameters of software components. This helps to standardize the notion of quality of software components by explicitly defining each of the included QoS parameters. It also aids in the acceptance of the underlying models for quantifying these parameters.

ii. A classification of the QoS parameters based on:

a. Domain of usage: Such a classification would enable a component user to identify the parameters that are of relevance to his/her domain.
b. Static/Dynamic behavior: Such a classification would be helpful to determine whether the value of a QoS parameter is constant or varies according to the environment. This would in turn help in determining whether the value of a QoS parameter can be improved by changes to the operating environment.
c. Nature of the parameter: Such a classification would help the component user to easily select all the parameters related to a specific aspect (like time-related or safety-related) of component quality of interest. The QoS parameters are classified according to their characteristics into: Time-related parameters (Turn-around-time), Importance–related parameters (priority), Capacity-related parameters (throughput, capacity), Integrity-related parameters (accuracy), Safety-related parameters (security) and Auxiliary parameters (portability, maintainability) as suggested in [OMG02].
d. Composability of the parameters: This kind of classification is of relevance when different components are integrated to form a software system. It indicates whether the value of a given QoS parameter can be used to arrive at the value of the corresponding QoS parameter of the resultant system. Some of the QoS parameters are inherently non-composable e.g.: parallelism constraints, priority, ordering constraints etc. Hence this kind of classification would prove to be valuable for a system integrator trying to determine the quality of an integrated system of components.

iii. By proposing standard approaches to account for the effect of the environment and the effect of usage patterns on the QoS of software components. These approaches involve an empirical evaluation of the effect of environment and the effect of usage patterns on the values of the QoS parameters of a software component, and a recording of these QoS values in tables. It is expected that, in practice, the component developers would perform these tests on their components and would then embed the resulting tables in the UniFrame descriptions of the components.

iv. By creation of an approach for investigation of the effects of component composition/decomposition on the QoS. This involves the development of composition/decomposition rules for estimating the QoS of an ensemble of software components given the QoS of individual components and to deduce the QoS of the individual components, given the QoS of the end-system.

The details regarding the QoS Catalog can be found in section 4.1. The details of the studies on the effect of environment and the effect of usage patterns are presented in section 4.2 and 4.3 respectively. The issue of effects of composition/decomposition on the QoS is not covered in this thesis, but is addressed in [SUN02].

In this chapter an overview of some of the work related to QoS in distributed systems was presented. This was followed by an analysis of the issues that arise when building a QoS framework for software components and the solutions offered to these issues by the UniFrame approach to QoS. In the next chapter, the details of the Unified Meta Model and the UniFrame Approach are presented, in order to put the UQOS framework and its realization into perspective, before proceeding into the details of the UQOS framework in chapter 4.

3. OVERVIEW OF THE UNIFRAME APPROACH (UA)

In the previous chapter an overview of some of the significant efforts in the area of QoS across various levels of a distributed computing system (DCS) was presented. In this chapter, the details of the Unified Meta model and the UniFrame Approach are given, along with an overview of the associated quality of service framework (UQOS). The UniFrame research is an attempt towards unification of the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques [BUR02]. The Unified Meta Model provides the theoretical foundation for the UniFrame Approach. [RAJ00], [RAJ01] and [RAJM01] provide more details about the UniFrame Approach and Unified Meta-component Model. The material in this chapter is based on these works.

3.1. Unified Meta-Component Model (UMM)

Unified Meta-component Model provides the foundation for implementing the UniFrame Approach. The core parts of UMM are as follows:

i. Components: The UniFrame Approach is component-based. Hence, components form the building blocks of any system built using the UniFrame Approach. Here, components are considered to be autonomous entities with non-uniform implementations. This means that the components may adhere to diverse distributed computing models. Every component has a state, an identity, a behavior, a well-defined interface and a private implementation. In addition to these parameters, every component has three aspects:

a. Computational Aspect: This refers to the task carried out by the component. It is a form of introspection by which every component describes its services to other components. UMM uses two categories of parameters namely:
    i. Inherent parameters: This consists of simple textual information containing the book-keeping information of a component.
    ii. Functional parameters: This consists of a formal and precise description of the computation, its associated contracts and the levels of service that the component offers.
b. Cooperative Aspect: This consists of,
    i. Pre-processing collaborators: other components on which this component depends, and
    ii. Post-processing collaborators: other components that may depend on this component
c. Auxiliary Aspect: This aspect addresses issues like mobility, security and fault tolerance of a component.

A sample natural language description of a UMM component, as depicted in [RAJ01] is shown in figure 3.1:

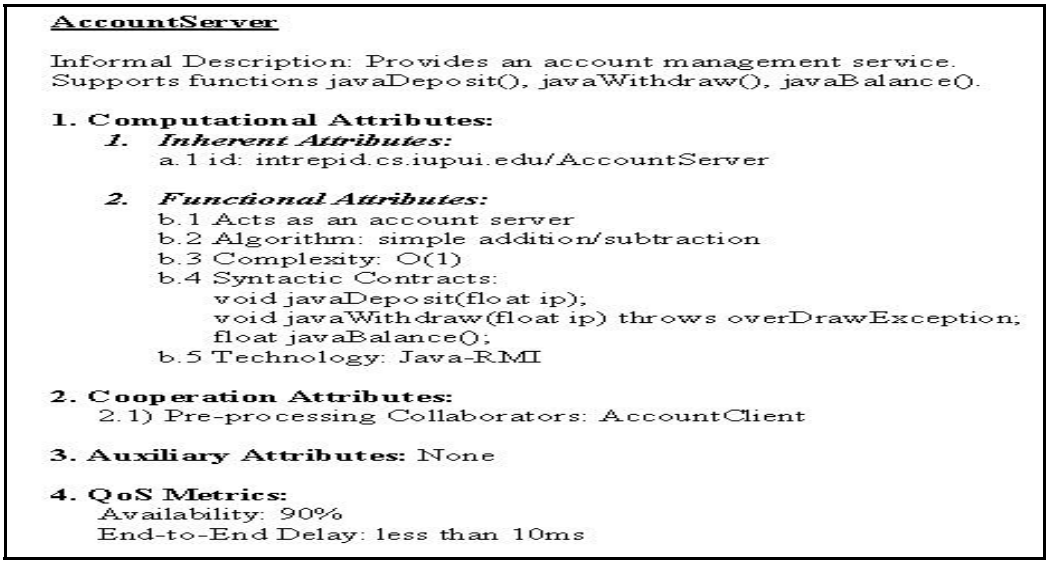

Figure 3.1. Informal Natural Language-based description of a UMM component

Component developers who wish to adopt the UniFrame Approach should specify the above mentioned parameters during the component development and deployment phase (dealt with in section 3.2.1). It is the responsibility of the component developer to ensure that his components meet the UMM specifications.

ii. Service and Service Guarantees: A UMM component offers services that may be in the form of an intensive computational effort or an access to underlying resources. The quality of the service offered by a component plays an important role in whether or not the component is selected for a given system. The quality of service of a component is an indication of the component developer's confidence in the ability of that component to carry out a specified service. In UMM, every component must specify the quality of service that it can offer in terms of the QoS Parameters, as identified in the QoS Catalog, which is described in section 4.1. The UQOS framework, which is the topic of this thesis, is an implementation of this aspect of the UMM and the UniFrame Approach. The details of the UQOS framework are presented in chapter 4.

iii. Infrastructure: UMM utilizes the *head-hunters* and the *Internet Component Brokers* (ICB) as infrastructure to address the issue of interoperability between heterogeneous DCS models. A brief introduction to the UMM infrastructure is provided below. The details of the infrastructure are dealt with in section 3.3.2.

The head-hunters are analogous to binders or traders in other models. The difference being that the trader is passive, with the components being responsible for registering themselves with the trader. On the other hand, the head-hunter actively discovers new components and attempts to register them with itself. A component may be registered with multiple head-hunters. It is also possible for multiple head-hunters to co-operate with each other in order to find a larger number of components.

The Internet Component Broker is intended to act as a mediator between components adhering to different component models. It utilizes adapter technology to provide translation capabilities between specific component architectures. The adapter components provide interoperability through *wrap* and *glue* technology [BER01]. The ICB is analogous to an Object Request Broker (ORB). The ORB provides the facilities for objects written in different programming languages to communicate, while the ICB provides the capability to generate glues and wrappers to allow components belonging to different component models to communicate.

In this section an overview of the core parts of the Unified Meta Model was presented. A more detailed look at these topics is presented in section 3.3.

3.2. The UniFrame Approach (UA)

"The UniFrame research attempts to unify the existing and emerging distributed component models under a common meta-model for the purpose of enabling discovery, interoperability, and collaboration of components via generative programming techniques. This research targets not only the dynamic assembly of distributed software systems from components built using different component models, but also the necessary instrumentation to enable QoS features of the component and the ensemble of components to be measured and validated using Event Grammars" [BUR02].

The QoS parameters, as identified in section 4.1, are broadly classified into static and dynamic parameters. The values of dynamic parameters change during run-time depending on the operating environment (CPU, Memory, process priority, etc), while the values of the static parameters do not change during run-time. In UniFrame, *Event Grammars* [AUG95, AUG97] are chosen as the system behavior model to measure and validate the dynamic QoS parameters. An *event* is defined as any detectable action performed during run-time, for instance, the execution of a statement or a call procedure. An event is associated with a beginning, end, duration and other parameters like the

program states at the beginning and end of an event and the source code associated with an event. There are two binary relations defined for events, one event may precede another event, or one event may be included in another. System execution is represented as a set of events with the two basic relations between them; this forms an '*event trace'*. An event grammar is a set of axioms that determines possible configurations of events of different types within the event trace.

The overview of the UniFrame Approach is illustrated in the figure 3.2 (graphics used with permission from [MIC02]).

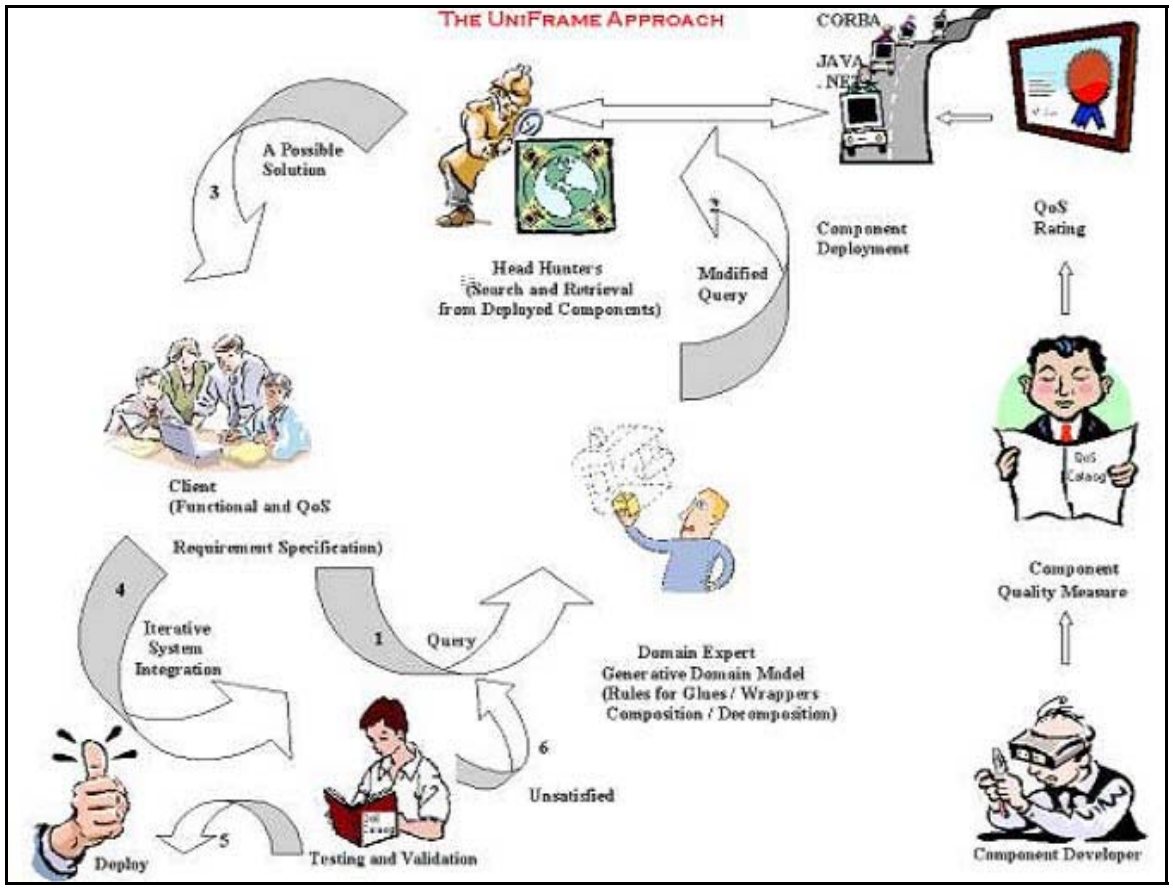

Figure 3.2. UniFrame Approach

The UniFrame approach to the creation of heterogeneous, component-based, distributed computing systems consists of the following phases [RAJ01]:

i. Component Development and deployment phase
ii. Automatic System Generation and QoS-based evaluation phase

3.2.1. Component Development and Deployment phase

This phase deals with the creation of the individual components that make up the end-system. Here, the component developers create the components and deploy them on a network to be found by the headhunters.

The UniFrame Approach is based on the Generative Programming [CZA00] paradigm. Here, it is assumed that the generation environment will be built around a generative domain specific model (GDM) supporting component-based assembly. This means that the components are created for a specific application domain, based on an accepted and standardized GDM.

The component development and deployment phase begins with the natural language-like specification of a component (as shown in section 3.3.1) and includes the computational, cooperative, and auxiliary aspects and QoS metrics of the component. The XML-based UniFrame specification (described in section 3.3.1) of this component is then derived from the natural language-like specification. The derivation process is based on the theory of Two-level Grammar (TLG) natural language specifications [BAR00, VAN65] and it is achieved by the use of conventional natural language processing techniques and a domain knowledge base. Generation of interfaces that include all the UniFrame aspects of the component is a part of this derivation process. The necessary implementations for the computational and behavioral methods are provided by the component developer. The Component developer then uses the QoS Catalog, dealt with in detail in section 4.1, to obtain the QoS parameters of the component. This is then followed by an empirical validation of the QoS of the component, which determines the values of the QoS parameters of the component. If the values of the QoS parameters are found to meet the QoS criteria entered in the specification, then the component is deemed to be ready for deployment. It is then deployed on the network to be discovered by the

headhunters. In case the QoS requirements are not met, the component developer either refines the UniFrame specification or the implementation, and the cycle is repeated.

The component development and deployment cycle as illustrated in [RAJM01] is shown in figure 3.3.

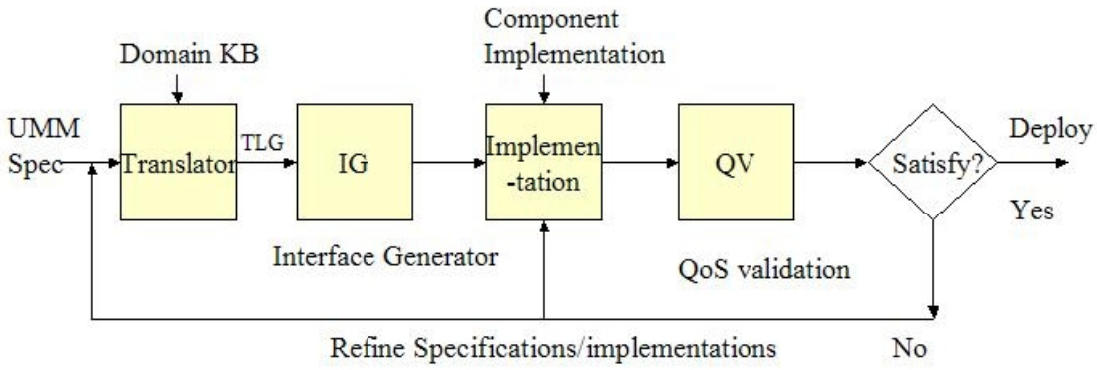

Figure 3.3. Component Development and Deployment Phase

3.2.2. The phase of Automatic System Generation of a system and its QoS-based Evaluation

The components created by the component developers in the component development and deployment phase are made available on a network to be located by the headhunters. Once the headhunters have located all the necessary components to build the system, the system assembly begins. The steps involved in this process are as follows:

i. The system developer who wishes to build a DCS presents a system query, in a structured form of natural language, which describes the required characteristics of the system. The Query Processor processes the query along with the help of the domain knowledge and a knowledge base containing the UniFrame description of the components for that domain. The output of the Query Processor is a formal specification, based on the theory of Two Level Grammar [BAR00, VAN65]. This formal specification of system

requirements is used by the headhunters for component searches and as an input to the system generation step. The GDM contains the domain knowledge such as the requirements specification and the matching design specification. The latter specifies the type of components required and the interdependence between these components. It also contains the composition/decomposition rules for the QoS parameters [SUN02]. Composition rules are intended for determining the QoS of the end-system given the QoS of the constituent components. The decomposition rules are intended for deducing the QoS of the constituent components given the QoS of the end-system. The Query processor uses the decomposition rules to deduce the QoS of the required components. It then creates a set of functional and QoS-based search parameters that serve as a guide to the headhunters to find the matching components in the search space.

ii. The headhunters collect a set of potential components meeting the functional and the QoS requirements from the given domain. The developer then selects a subset of components from this set based on their QoS values, setting aside the remaining components for later consideration (if necessary).

iii. The subset of components returned may or may not contain all the required components satisfying the functional and the QoS requirements. In the latter case, the process may request additional components or attempt to refine the system query by adding more information about the desired solution from the problem domain. In case all the required components are found, a system is now assembled by the System Generator using the selected components according to the generation rules embedded in the system design specification. These components in combination with the appropriate adapters (if needed) form a software implementation of the target system. The adapter components provide interoperability between component models through *wrap* and *glue* technology [BER01]

iv. This implementation is now tested using event traces and a set of test cases to verify that it meets the desired functional and QoS criteria. In case it does not,

it is discarded and another implementation of the target system is built from the component collection. This process is repeated until an optimal (with respect to the QoS) implementation is built, or until the component collection is exhausted. In case the collection is exhausted, additional components are requested or the initial query is refined by adding more information about the desired solution from the problem domain. This process continues as long as it takes to build the required DCS satisfying the functional and QoS requirements or until the system developer is satisfied with the generated system. Upon creation of a satisfactory implementation, the system is deemed ready for deployment.

The figure below indicates the process for automated system generation and evaluation as depicted in [RAJM01]:

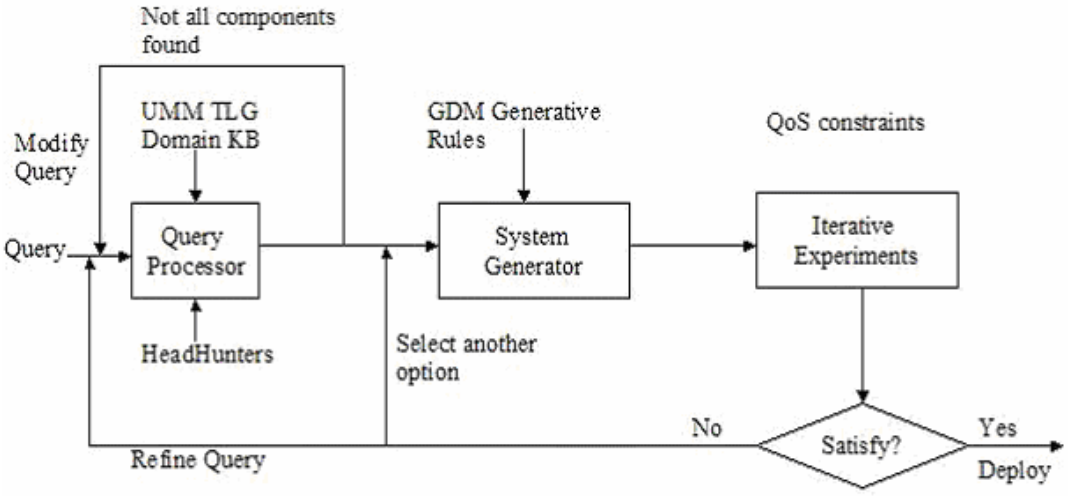

Figure 3.4. Automated System Generation and Evaluation

3.3. Details of the Unified Meta Model (UMM)

An overview of the Unified Meta Model was presented in section 3.1. In this section, more detailed descriptions are presented of two of the core parts of UMM, namely, the components and their specification, and the infrastructure. The remaining core part, the Service and service guarantees is described in Section 3.4.

3.3.1. Specification of Components in the UMM

In UniFrame, the component specification is initially implemented in natural language. This natural language specification contains the computational, cooperative, auxiliary parameters and the QoS metrics for the given component. A sample natural language specification of a component in UniFrame is shown in figure 3.5.

1. Name: BankClient
2. Domain: Banking
3. Informal Description: Requests account services from an appropriate server and interact with users.
4. Computational Attributes:
    - 4.1 Inherent Attributes:
        - 4.1.1 id: magellan.cs.iupui.edu:1099/BankClient
        - 4.1.2 Version: 1.0
        - 4.1.3 Author: DCS Lab
        - 4.1.4 Date: 10/8/2002
        - 4.1.5 Validity: 6 months
        - 4.1.6 Atomicity: Yes
        - 4.1.7 Registration: www.cs.iupui.edu/headhunter1
        - 4.1.8 Model: Java RMI 1.3.1
    - 4.2 Functional Attributes
        - 4.2.1 Function description: Accept user queries and presents the results using GUI.
        - 4.2.2 Algorithm: Java Foundation Classes(JFC)
        - 4.2.3 Complexity: 0(1)
        - 4.2.4 Syntactic Contract:
            - void withdraw(double amount);
            - void deposit(double amount);
            - double checkBlance();
        - 4.2.5 Technology: Java RMI
        - 4.2.6 Preconditions:
        - 4.2.7 Postconditions:
        - 4.2.8 Invariant:
        - 4.2.9 Expected Resources: N/A
        - 4.2.10 Design Patterns: NONE
        - 4.2.11 Known Usage: NONE
        - 4.2.12 Alias: NONE
5. Cooperation Attributes:
    - 5.1 Preprocessing Collaborators: ***NONE***
    - 5.2 Postprocessing Collaborators: AccountServer, AccountManager
6. Auxiliary Attributes:
    - 6.1 Mobility: ***No***
    - 6.2 Security: ***L0***
    - 6.3 Fault tolerance: ***L0***
7. QoS Metrics: throughput, end-to-end delay
8. QoS Level: ***L1***
9. Cost: ***L1***
10. Quality Level: ***L2***

Figure 3.5. Example of Informal Natural Language-based UniFrame Specification

During the Component Development and deployment phase, the natural language specification is converted into a standardized XML-based specification. The components of the XML specification are:

i. ID: This is a unique string consisting of the host name and the port on which the component is running along with the name with which the component binds itself to a registry. Example: Intrepid.cs.iupui.edu:8080/AccountServer

ii. Component Name: This is the name used by the component to identify itself. This can be different from the name used in the ID as stated earlier.
Example: AccountServer

iii. Description: This is a brief description of the service provided by the component.
Example: Provides an account management system.

iv. Function Descriptions: This provides a brief description of each of the functions supported by the component.
Example: javaDeposit: provides a deposit service for a savings account, javaWithdraw: provides a withdrawal service for a savings account, javaBalance: provides a balance checking service for a savings account .

v. Syntactic Contracts: This provides the computational signature of the component's service interface.
Example: *void javaDeposit(float ip), void javaWithdraw(float ip), void javaBalance( )*

vi. Purpose: provides a description of the overall functionality of the component.
Example: acts as an account server

vii. Algorithm: Indicates the algorithms utilized by the component to implement its functionality.
Example: Simple addition and subtraction

viii. Complexity: Describes the order of complexity of the above mentioned algorithms implemented by the component.
Example: O(1)

ix. Technology: Indicates the component technology utilized to implement the component.
Example: J2EE, CORBA, .NET etc

x. QoS Metrics: Indicates the values for the QoS parameters of the component as specified by the manufacturer of the component. It is represented as the triplet < QoS parameter name, measure, value> where, QoS parameter name may be one of various QoS parameter names like throughput, dependability, and capacity. Measure indicates the unit of measure used to quantify the QoS parameter like results per second or number of concurrent requests per second. Value indicates the actual measured numeric value (or range) of the QoS parameter for the component. The example below illustrates a natural language-like specification for a Java-RMI based savings account management system with facilities for account balance check, deposit and withdraw.
Example: Availability: greater than 90%, End-to-End Delay: less than 10 ms

The natural-language like specification presented in figure 3.5 can be translated into the following XML-based UniFrame specification, as illustrated in [SIR02].

```
<UniFrame>

    <ComponentName> AccountServer </ComponentName>
    <Description> Provides an Account Management System </Description>

    <FunctionDescription>
        <Function> javaDeposit </Function>
        <Function>javaWithdraw </Function>
        <Function> javaBalance </Function>
    </FunctionDescription>

    <ComputationalAttributes>
        <InherentAttributes>
            <ID> intrepid.cs.iupui.edu/AccountServer </ID>
        </InherentAttributes>

        <FunctionalAttributes>
            <Purpose> Acts as Account Server </Purpose>
            <Algorithm> Simple Addition/Subtraction </Algorithm>
            <Complexity> O(1) </Complexity>
            <SyntacticContract>
                <Contract> void javaDeposit(float ip) </Contract>
                <Contract>
                    void javaWithdraw throws OverDrawException
                </Contract>
                <Contract> float javaBalance() </Contract>
            </SyntacticContract>
            <Technology> Java-RMI </Technology>
        </FunctionalAttributes>
    </ComputationalAttributes>

    <CooperatingAttributes>
        <PreprocessingCollaborators> AccountClient </PreprocessingCollaborators>
    </CooperatingAttributes>

    <AuxillaryAttributes>
        <Mobility> No </Mobility>
    </AuxillaryAttributes>

    <QOSMetrics>
        <Availability measure="%"> 90 </Availability>
        <End2EndDelay measure="ms" > 10 </End2EndDelay>
    </QOSMetrics>

</UniFrame>
```

Figure 3.6. Example of Translated XML-based UniFrame Specification

3.3.2. Infrastructure

An overview of the UMM Infrastructure was provided in section 3.1. Here, the details of the UniFrame Discovery Service (URDS) [SIR02], which is an implementation of the UMM infrastructure, are provided. An illustration of the URDS architecture as depicted in [SIR02] is shown in figure 3.7 (reproduced from [SIR02] with permission).

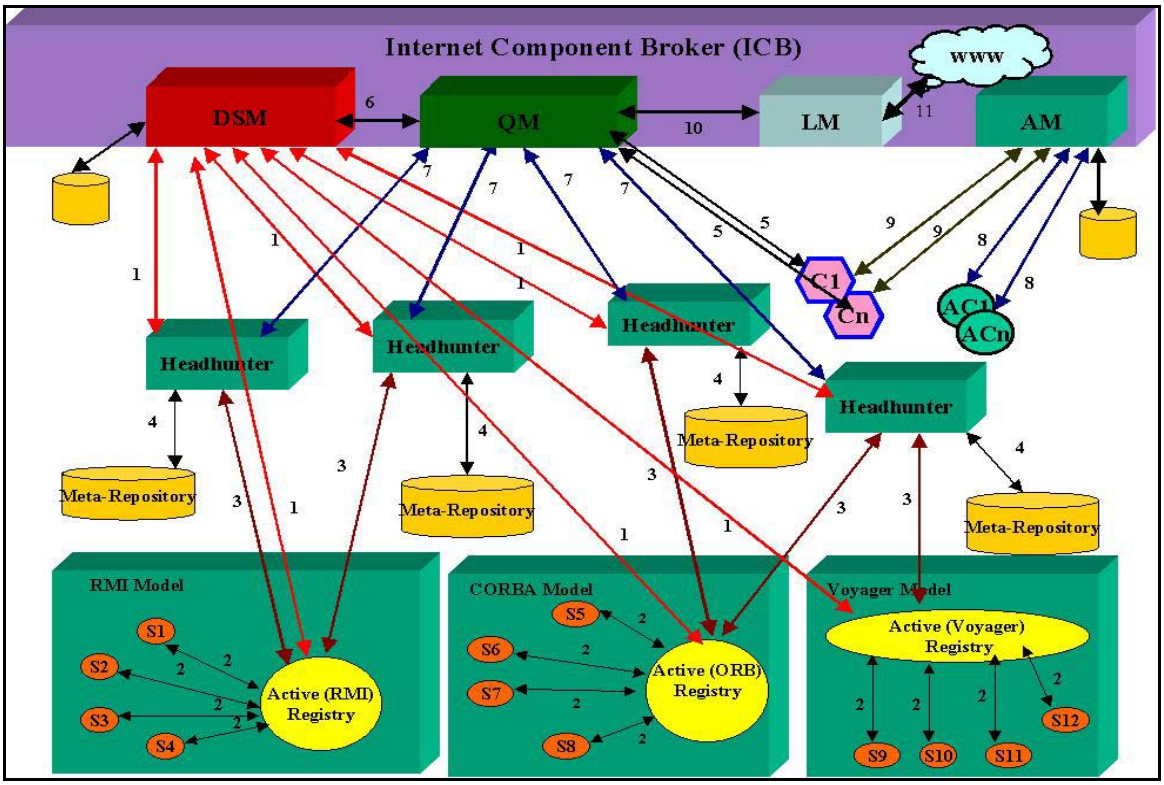

Figure 3.7. URDS Architecture

The URDS infrastructure comprises the following components:

i. Internet Component Broker (ICB): It is a collection of the following services - Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM). It acts as an all-pervasive component broker in an interconnected environment. The communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between heterogeneous components are all contained in the ICB. The constituent services of ICB are all accessible at well-known addresses. It is anticipated that there will be a fixed number of ICBs deployed at well-known locations hosted by organizations supporting the UniFrame Approach.

   a. Query Manager (QM): The QM is used to translate a system integrator's requirements specification for a component (dealt with in section 3.3.1) into a Structured Query Language (SQL) statement and dispatch this query to the appropriate head-hunters. The headhunters, in turn, return lists of

service provider components that match the search criteria contained in the query. The QM and the Link Manager together are responsible for propagating the queries to other linked ICBs.

b. Domain Security Manager (DSM): The URDS discovery protocol is based on periodic multicast announcements. The multicasting exposes the URDS to a number of security threats. The DSM is responsible for ensuring that the security and integrity of the URDS are maintained. The security scheme implemented by the DSM involves the generation and distribution of secret keys for the ICB. It also enforces multicast group memberships and controls access to multicast addresses allocated for a particular domain, through authentication and use of Access Control Lists. Access Control Lists allow a sender or an authorized third party to maintain an inclusion or an exclusion list of hosts on the Internet corresponding to a multicast group. Each time a host requests to join the multicast group, the sender or the third party checks with the access control list to determine whether the host is authorized to join the group.

c. Link Manager (LM): It establishes links between ICBs to form a federation and propagate the queries received from the QM to the linked ICBs. The ICB administrator configures the LM with the location information of LMs of other ICBs with which links are to be established.

d. Adapter Manager: It acts as registry or lookup service for clients seeking adapter components. The adapter components register with the AM and at the same time indicate which component models they can bridge efficiently. The AM is contacted by the clients to locate the adapter components matching their requirements.

ii. Headhunters (HH): The responsibilities of the headhunter include: detection of presence of service providers (service discovery), registration of functionality of the service providers and returning to the ICB a list of discovered service providers that match the requirements.

iii. Meta-Repository (MR): It is a database that serves a headhunter by holding the UniFrame specification information of exporters. In URDS the MR is implemented presently as a relational database using Oracle.

iv. Active-Registries (ARs): These listen and respond to multicast messages from headhunters. Each also has introspection capabilities to discover not only the instances, but also the specifications of the components registered with them. URDS implements them by extending the native registries or lookup services of component models like RMI, CORBA and Voyager.

v. Services (S1..Sn): The services may be implemented in diverse component models. Each identifies itself by the service type name and the XML description of the component's informal UniFrame specification containing the computational, functional, co-operational and auxiliary parameters, and QoS metrics for the component.

vi. Adapter Components (AC1..ACn): They serve as bridges between components implemented in different component models like (J2EE, CORBA, .NET).

In this section, overviews of the UMM component specification and the UMM infrastructure were presented. In the next section, a brief introduction to the UQOS framework and its objectives is given.

3.4. Overview and Objectives of the UQOS Framework

As explained in section 3.1, Service and Service guarantees are an integral part of every component in UMM and they also play an important role in the system generation phase of the UniFrame approach. The UQOS framework is an implementation of the Service and Service guarantees aspect of the UMM and the UniFrame Approach.

In order to utilize the Service and Service guarantees of UMM in a real-world CBSD scenario, there are a few issues that need to be addressed. A brief introduction to these issues is provided here, more details can be found in chapter 4. The first issue is the lack of standardization within the software community regarding the quality of software components. Also, according to the CBSD philosophy, a given component may be used under diverse operating environments (CPU, memory, operating system and priority schemes) and usage patterns (the pattern of users and user requests received by components), which can affect the Quality of Service (QoS) offered by the software component. Several interface definition languages exist to specify the functional aspects of a component. Along similar lines a specification scheme is required to express the QoS aspects of a software component. This calls for an objective paradigm for quantifying and specifying the quality of software components, as well as accounting for the effect of the environment and the effect of usage patterns on the QoS of software components. UQOS framework is designed to address these issues.

The objectives of the UQOS framework can be stated as follows:

- To act as a framework to objectively quantify the QoS of software components.

- To standardize the notion of quality of software components by using the QoS Catalog and to make the software component developers (producers) and system integrators (consumers) use the QoS Catalog as a reference guide.

- To provide a standard approach to incorporate the effect of the environment on the QoS of software components into the component development process.

- To provide a standard approach to incorporate the effect of usage patterns on the QoS of software components into the component development process.

- To provide a QoS specification scheme to specify the QoS of software components.

To facilitate the realization of the above mentioned objectives, the UQOS framework has been partitioned into four major parts, namely, *the QoS Catalog*, *the approach for accounting for the effect of the environment on the QoS of software components*, *the approach for accounting for the effect of usage patterns on the QoS of software components* and *the specification of the QoS of software components*. The QoS Catalog is intended to act as a tool for standardizing the notion of Quality of software components. It contains detailed descriptions of QoS parameters of software components, including the metrics, the evaluation methodologies, the factors influencing these parameters and the interrelationships among these parameters. The approaches for accounting for the effect of the environment and the effect of usage patterns on the QoS of software components consist of an empirical validation of the QoS of the software components under diverse environmental conditions and usage patterns. The resulting QoS values are then specified in the component interface. An in-depth look at the various parts of the UQOS framework is provided in chapter 4.

In this chapter, an overview of the UniFrame Approach, along with the associated Unified Meta Model, was presented. The chapter also presented a brief introduction to the UQOS framework. In the following chapter, the details of the UQOS framework are presented, along with a discussion of how the above stated objectives are achieved by the UQOS framework.

4. IMPLEMENTATION OF THE UQOS FRAMEWORK

In the previous chapter, overviews of the Unified Meta Model, the UniFrame Approach and the UQOS framework were presented. In this chapter, an in-depth look at the various parts of the UQOS framework is presented. As stated in chapter 3, the UQOS framework consists of four main parts, namely: the QoS Catalog, the approach for accounting for the effects of environment on the QoS of software components, the approach for accounting for the effects of usage patterns on the QoS of software components and the specification of the QoS of software components. This chapter begins with a look at the QoS Catalog for Software components, then the details of the work on the effects of environment and the effects of usage patterns on the QoS of software components is presented, and in the end, the issue of specification of the QoS of software components is addressed.

4.1. Quality of Service (QoS) Catalog for Software Components

The creation of a QoS Catalog is the first step in an effort to build the UQOS framework. One of the primary hurdles to the creation of a QoS framework, such as the UQOS framework, is the general disagreement among component developers regarding what constitutes the "Quality" of a software component and the techniques used to measure quality. The QoS Catalog is primarily intended to address these issues and to standardize the notion of software component quality. The QoS Catalog overcomes these hurdles by providing:

- A compilation of QoS parameters, along with their definitions.

- A classification of these parameters based on different criteria, such as, domain of usage, static or dynamic behavior, nature of the parameters and the Composability of the parameters.
- An incorporation of the methodologies for quantifying each of the QoS parameters.

4.1.1. Motivation for the Catalog

The QoS Catalog is intended to act as a tool for standardization of the notion of Quality of software components. It is designed to act as a comprehensive source of information regarding the quality of software components. The comprehensiveness of the catalog can be guaranteed by the possibility of an evolution to support the discovery of new parameters and quantification methodologies. The catalog contains detailed descriptions about the QoS parameters of software components including the metrics, evaluation methodologies, the factors influencing the parameters and the interrelationships among the parameters. The QoS Catalog, used in conjunction with the UniFrame approach, would force the component developer to consider and validate the QoS of a component before advertising its quality. The motivation for creating the QoS Catalog is two fold; it would prove to be a valuable tool for

a) The component developer, by:
    - Acting as a reference manual for incorporating QoS parameters into the components being developed.
    - Allowing him to enhance the performance of his components in an iterative fashion by being able to quantify their QoS parameters.
    - Enabling him to advertise the Quality of his components, after validation, by utilizing the QoS metrics.

b) The System Developer, by:
    a. Enabling him to specify the QoS requirements for the components that are incorporated into his system.

b. Allowing him to verify and validate the claims made by a component developer regarding the quality of a component before incorporating it into the system.
c. Allowing him to make objective comparisons of the Quality of Components having the same functionality.
d. Empowering him with the means to choose the best-suited components for his system.

In January 2002 the Object Management Group (OMG) issued an RFP (Request For Proposal) for a "UML profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" [OMG02]. This is seen as a concrete step towards incorporating QoS characteristics into the CORBA framework. It is believed that the UQOS framework provides solutions to some of the issues set forth in [OMG02].

The key mandatory requirements of [OMG02] are shown in figure 4.1.

A General Quality of Service Framework
To ensure consistency in modeling various qualities of service, submissions shall define a standard framework or, reference model, for QoS modeling in the context of the UML. This shall include:

- A general categorization of different kinds of QoS; including QoS that are fixed at design time as well as ones that are managed dynamically
- Integration of different categories of QoS for the purpose of QoS modeling of system aspects.
- Identification of the basic conceptual elements involved in QoS and their mutual relationships. This shall include the ability to associate QoS characteristics to model elements (specification), a generic model of the system aspects involved in QoS-associated collaboration and their functional interactions and use cases (usage model), and a generic model of how QoS allocation and decomposition is managed.
- A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

A Definition of Individual QoS Characteristics
Submissions shall define QoS characteristics, particularly those important to real-time and high confidence systems, which describe the fundamental aspects of the various specific kinds of QoS based on the QoS categorization identified in the framework. These shall include but are not limited to the following:

- time-related characteristics (delays, freshness)
- importance-related characteristics (priority, precedence)
- capacity-related characteristics (throughput, capacity)
- integrity related characteristics (accuracy)
- fault tolerance characteristics (mean-time between failures, mean-time to repair, number of replicas)

A coherent set of stereotypes, tagged values, and constraints as necessary to represent the identified QoS properties constructing a UML Profile.

Figure 4.1. Key Mandatory Requirements of OMG RFP for UML Profile for QoS

As seen in the figure 4.1., the OMG RFP calls for a general classification of different kinds of QoS, including QoS that is fixed at design time as well as ones that are dynamically managed. The QoS Catalog satisfies this requirement by classifying the QoS parameters as static (fixed at design-time or implementation time) and dynamic (varying at run-time, depending on the operating environment and the usage patterns).

Further, the RFP requires definitions of different QoS characteristics (called parameters in the QoS Catalog) and their classification into time-related characteristics, importance-related characteristics, capacity-related characteristics, integrity-related characteristics and fault tolerance characteristics. The QoS Catalog incorporates a classification scheme corresponding to this and builds upon it by a classification based on the domain of usage of the parameters and the composability of the parameters (described in section 4.1.2.).

4.1.2. Objectives of the Catalog

The objectives of the QoS Catalog are as follows:

a) Identification of QoS parameters of software components: The objective is to prepare a list of QoS parameters which would act as a checklist for any component developer interested in using QoS parameters in his components.

b) Classification of QoS parameters based on:
    i. Domain of usage: Such a classification would enable a component user to identify the parameters that are of relevance to his/her domain.
    ii. Static/Dynamic behavior: Such a classification would be helpful to determine whether the value of a QoS parameter is constant or varies according to the operating environment and the usage patterns. This would in turn help in determining whether the value of a QoS parameter can be improved by changes to the operating environment and the usage patterns.

iii. Nature of the parameter: The QoS parameters identified are classified according to their characteristics into: Time-related parameters (Turn-around-time, throughput), Importance–related parameters (priority), Capacity-related parameters (capacity), Integrity-related parameters (accuracy, precision), Fault Tolerance-related parameters (dependability) and Auxiliary parameters (portability, maintainability) as suggested in [OMG02].

iv. Composability of the parameters: This kind of classification is of relevance when different components are integrated to form a software system. It indicates whether the value of a given QoS parameter can be used to arrive at the value of the corresponding QoS parameter of the resultant system. Some of the QoS parameters are inherently non-composable for example, parallelism constraints, priority, ordering constraints. Hence, this kind of a classification would prove to be valuable for a system integrator trying to determine the quality of an integrated system of components.

c) Incorporation of methodologies for quantification of QoS of software components based on their: Reproducibility (ability to produce consistent values in different trials), Objectivity (the fairness and impartiality of the methodology), Precision (the level of detail or granularity provided) and suitability for the component-based framework. This helps to create uniformity in the quantification of the QoS.

d) Identification of the factors influencing each of the identified QoS parameters. This helps to improve a particular parameter by varying the factors influencing it.

e) Identification of interrelationships between the QoS parameters. It is possible that a variation in one QoS parameter may also affect other parameters. This helps to identify those parameters that are affected by or affecting the given parameter.

The QoS Catalog presented here, as a part of the UQOS framework, satisfies the objectives stated above as follows:

1. By inclusion of some of the major QoS parameters used across various domains in the industry, into the catalog. However, the list of included parameters is not comprehensive. This is because, it is felt that the inclusion of QoS parameters into the catalog should be an evolutionary process (rather than a static effort). Consequently, a provision for the versioning of the catalog has been incorporated to deal with the evolution of the catalog. As a means to enforce the versioning mechanism, it is required that the UMM description of a component should clearly state the version of the catalog it conforms to.
2. By classifying each of the included parameters on the basis of their
    a. Domains of usage, the domains considered are based on the OMG domain task force groups [DTF00]. These domains are, C4I (Command, Control, Communication, Intelligence), Finance, Healthcare, Life Sciences, Manufacturing, Space, Telecom, Transportation, E-Commerce, Real-time, and Utilities (gas, electric).
    b. Static or dynamic behavior.
    c. Nature (according to [OMG02]) and
    d. Composability.
3. By incorporating measurement models, for quantifying each of the included parameters. For those parameters which do not have established measurement models, heuristic models have been temporarily adopted, pending development of more comprehensive models. This is plausible, due to the versioning mechanism incorporated into the catalog.
4. By including for each parameter in the catalog, the factors on which the parameter depends on. These factors are determined by the semantics of the specific measurement model used to quantify the parameter.
5. By identifying the interrelationships between the parameters included in the catalog. This is achieved by indicating for a given parameter, those parameters that might affect the parameter and those parameters affected by the parameter.

For a better interpretation of the information presented in this section, the format of the QoS Catalog, along with brief descriptions of the contents of the catalog, are presented in section 4.1.3.

4.1.3. Format of the Catalog

The general format used to describe each parameter in the catalog is outlined below. This format is based on the design patterns catalog [GAM95].

1. Name: Indicates the name of the parameter.

2. Intent: Indicates the purpose of the parameter.

3. Description: Provides a brief description of the parameter.

4. Motivation: States the motivation behind the inclusion of the parameter and the importance of the parameter.

5. Applicability: Indicates the type of systems where the parameter can be used.

6. Model Used: Indicates the model used for Quantification of the parameter.

7. Influencing Factors: Indicates the factors on which the parameter depends.

8. Measure: Indicates the unit used to measure the parameter.

9. Evaluation Procedure: Outlines the steps involved in the quantification procedure.

10. Evaluation Formulae: Indicates the formulae used in the evaluation procedure.

11. Result Type: Indicates the type of the result returned by the evaluation procedure.

12. Nature: Indicates the nature of the parameter as suggested in [OMG02].

13. Static/Dynamic: Indicates whether the value of the parameter is constant or varies during run-time.

14. Increasing/Decreasing: Indicates whether higher values of the parameter correspond to better QoS (Increasing) or lower values correspond to better QoS (Decreasing).

15. Composable/Non-composable: Indicates whether the parameter can be used during the component composition process to arrive at the QoS value of the end-system using the QoS values of the individual components.

16. Consequences: Indicates the possible effects of using the chosen model to quantify the parameter.

17. Related Parameters: Indicates the other related QoS parameters.

18. Domain of Usage: Indicates the domains where the parameter is widely used.

19. User Caution: It warns the user about the consequences of choosing a component with a lower level of a QoS parameter over another component (having the same functionality) with a higher level of the QoS parameter.

20. Aliases: Indicates other prevalent equivalent names for a parameter, if any.

4.1.4. Parameters included in the catalog

In this section, the parameters currently included in the catalog, along with their brief descriptions are presented. More parameters shall be included as the catalog evolves. Some of the parameters, namely, parallelism constraints, ordering constraints, achievability and priority have been selected from [LOY98].

1. *Dependability:* It is a measure of confidence that the component is free from errors.

2. *Security:* It is a measure of the ability of the component to resist an intrusion.

3. *Adaptability:* It is a measure of the ability of the component to tolerate changes in resources and user requirements.

4. *Maintainability:* It is a measure of the ease with which a software component can be maintained.

5. *Portability:* It is a measure of the ease with which a component can be migrated to a new environment.

6. *Throughput:* It indicates the efficiency or speed of a component.

7. *Capacity:* It indicates the maximum number of concurrent requests a component can serve.

8. *Turn-around Time:* It is a measure of the time taken by the component to return the result.

9. *Parallelism Constraints:* It indicates whether a component can support synchronous or asynchronous invocations.

10. *Availability:* It indicates the duration when a component is available to offer a particular service.

11. *Ordering Constraints:* It indicates whether the results returned by a component are in the proper order.

12. *Priority Mechanism:* It indicates if a component is capable of providing prioritized service.

Detailed descriptions of two of the above mentioned parameters, Dependability (static parameter) and Turn-around-time (dynamic parameter) are provided below. The detailed descriptions of all the parameters can be found in [BRA01].

Table 4.1. Description of Dependability

DEPENDABILITY	
Intent:	It is a measure of confidence that the component is free from errors.
Description:	It is defined as the probability that the component is defect free.
Motivation:	1. It allows an evaluation of degree of Dependability of a given component. 2. It allows Dependability of different components to be compared. 3. It allows the possibility of modifications to a component to increase its Dependability.
Applicability:	This model can be used in any system, which requires its components to offer a specific level of dependability. Using the model, the Dependability of a given component can be calculated before being incorporated into the system.
Model Used:	Dependability model by Jeffrey Voas [VOA95], [VOA98], [VOA00].

Metrics used: Testability Score, Dependability Score.

Influencing Factors:

1. Degree of testing.
2. Fault hiding ability of the code.
3. The likelihood that a statement in a component is executed.
4. The likelihood that a mutated statement will infect the component's state.
5. The likelihood that a corrupted state will propagate and cause the component output to be mutated.

Evaluation Procedure:

1. Perform Execution Analysis on the component to find the execution estimate, which is the probability of executing a particular statement in the component.
2. Perform Propagation Analysis on the component to find the propagation estimate, which is the probability that a data state produced by a statement has an effect on the component output.
3. Calculate the Testability score for each statement of the component, which is a prediction of the likelihood that the statement will hide a defect during testing. The lowest testability score of any statement in the component is now selected as the Testability score of the component.
4. Calculate the Dependability Score of the Component.

Evaluation Formulae:

$T_i = E_i * P_i$

T_i: Testability Score for statement 'i'

E_i: Execution Estimate for statement 'i'

P_i: Propagation Estimate for statement 'i'

$T = \min (T_i)$

T: Testability Score of the component

$D = 1-(1-T)^N$

D: Dependability Score.

N: Number of successful tests.

Result Type:	Floating Point Value between [0,1]
Nature:	Fault Tolerance-related.
Static/Dynamic:	Static
Increasing/Decreasing:	Increasing
Composable/Non-Composable:	Composable.
Consequence:	1.Greater Testability scores result in greater Dependability. 2.Lower Testability scores result in lesser Dependability. 3.Lesser amount of testing is required to provide a fixed dependability score for higher Testability Scores. 4.Additional testing can improve a poor dependability score.
Related Parameters:	Security, Availability
Domain of Usage:	Domain Independent
User Caution:	Lower dependability may result in: 1.Higher chances of unreliable component behavior. 2.Higher possibility of improper execution/termination. 3.Higher possibility of erroneous results.
Aliases:	Maturity, Fault Hiding Ability, Degree of Testing

Table 4.2. Description of Turn-around-time

TURN-AROUND-TIME	
Intent:	It is a measure of the time taken by the component to return the result.
Description:	It is defined as the time interval between the instant the component receives a request until the final result is generated.
Motivation:	1. It indicates the delay involved in getting results from a component. 2. It is one of the measures of component performance.
Applicability:	This attribute can be used in any system, which specifies bounds on the response times of its components.
Model Used:	Empirical approach.
Metrics Used:	Mean Turn-around-time.
Influencing Factors:	1. Implementation (algorithm used, multi-thread mechanism etc). 2. Speed of the CPU. 3. Available memory. 4. Process priority. 5. Usage Pattern. 6. Computer Organization. 7. Hardware resources like floating point processor, system bus, I/O devices,etc. 8. Operating System's access policy for resources like: CPU, I/O, memory, etc.

Evaluation Procedure:	1. Record the time instant at which the request is received. 2. Record the time instant at which the final result is produced. 3. Repeat steps 1 and 2 for 'n' representative requests. 4. Calculate the Mean Turn-around Time.
Evaluation Formulae:	MTAT= $[\sum_{i=1}^{n} (t2-t1)] / n$. Where, *MTAT: Mean Turn-around Time.* *t1: time instant at which the request is received.* *t2: time instant at which the final result is produced.* *n: number of representative requests.*
Result Type:	Floating Point Value in milliseconds.
Nature:	Time-related.
Static/Dynamic:	Dynamic.
Increasing/Decreasing:	Decreasing.
Composable / Non-Composable	Composable.
Consequence:	The lower the time interval between the instant the request is received and the response is generated, the lower the Mean Turn-around Time.
Related Parameters:	Throughput, Capacity.
Domain of Usage:	Domain Independent.

User Caution: A higher value of Turn-around-time results in:
1.Longer delays in producing the result.
2.Higher round trip time.

Aliases: Latency, Delay.

4.2. Effect of environment on the QoS of Software Components

This section deals with the aspect of the UQOS framework that involves the study of the effect of environment on the QoS of software components. Presented in this section, are the motivation, the objectives and the approach used by the UQOS framework to study the effect of environment on the QoS of software components.

4.2.1. Motivation

A brief introduction to the issue of the effect of environment on the QoS of software components was given in section 2.5. Here, a more detailed look at the motivation behind this study is presented.

According to the CBSD philosophy, a given component may be used under diverse environments. The definition of environment here includes those features (called *environment variables*) of the execution platform of a software component, which might have a significant impact on the QoS of that component. Some of these environment variables are: the CPU speed, the memory, the process priority assigned to the component, the operating system used etc. The fact that the environment variables can affect the QoS of a software component implies that any QoS associated with a software component would not necessarily hold true in foreign environments. Hence, it becomes critical to account for the effect of the execution environment on the QoS of software components.

The other factor motivating this study is the possibility of enhancing the QoS of a software component by suitably varying its execution environment. A component

user might desire to improve a component's QoS (depending on the component's semantics) by suitably altering its execution environment (like providing a faster processor, increasing the memory etc). The information related to the effect of the environment on the given component would act as a valuable guide to the component user involved in this activity, by providing information about the variation in the QoS that can be expected with a corresponding change in the value of each environment variable.

4.2.2. Objectives

The objectives of the study on the effect of environment on the QoS of software components are:

- To create a standardized approach to account for the effect of the environment on the QoS of software components.
- To make the effect of the environment on QoS an integral part of the UniFrame approach.
- To force a component developer, adhering to the UniFrame approach, to consider the effect of environment on the QoS of his/her components.
- To provide the component user with a means to deduce the QoS of a given component under the specified environment conditions.
- To act as a guide to the component user interested in enhancing the QoS of a component by altering its execution environment.

4.2.3. Approach

The UQOS framework prescribes a specific set of steps to be followed by a component developer, adhering to the UniFrame approach, in order to account for the effect of environment on the QoS of his components. The approach prescribed by the UQOS framework, for a component developer interested in developing a software component with a specific functionality, for a given domain and adhering to the domain model for the chosen domain, is as follows:

1. To prepare a list of QoS parameters of relevance to the chosen domain based on the QoS Catalog.
2. To create/incorporate the QoS instrumentation code for each of the chosen parameters, adopting the QoS quantification models prescribed in the QoS Catalog.
3. To select the set of environment variables of relevance as defined in the domain model for the component.
4. For each selected parameter P_i (i=1 to n),
    a. If P_i is static,
        i. for each set of representative test cases, t_c (c=1 to n)
        Run the instrumentation code, record the values
        ii. Include the QoS metrics in the UniFrame description of the component.
    b. Else, If P_i is dynamic,
    Vary the set ofenvironment variables E_j (j=1 to m) as follows:
    Select a subset E_s (s=1 to k) of E_j
        i. Vary the environment variables in the subset E_s while keeping the variables in the set (E_j- E_s) constant.
        ii. Run the instrumentation code and record the value of the parameter P_i for each set of values of environment variables in E_s.
        iii. Plot a graph of P_i versus E_s.
        iv. Prepare a table with values of E_s and P_i.

v. Include the prepared table in the UniFrame description of the component.

5. Proceed to the effect of usage patterns (described in section 4.3.).

The approach presented above satisfies the objectives stated in section 4.2.2. as follows:

1. The above stated approach ensures that the effect of environment on the QoS of software components is considered, by making it an integral part of the UniFrame component development process.
2. By the inclusion of the effect of environment information in the UniFrame description of components, the component developer is forced to account for the effect of environment, in order to comply with the UniFrame approach. The component developer's desire to compete with other component developers complying with the UniFrame guidelines acts as another incentive to persuade him to account for the effect of environment on the QoS of his components.
3. A component user interested in finding the QoS of a component in his own environment, would be most likely able to deduce that information from the table included in the UniFrame description of the component, containing the values of each dynamic QoS parameter against different environment variables.
4. A component user interested in enhancing the QoS of a component can infer from the table, the possible gains in the QoS values that could be achieved (provided the licensing agreement between the user and the developer allows this) by varying the environment variables. This helps him to decide upon which environment variable to change and by how-much, in order to realize the required enhancement in the QoS of the component.

This section presented an overview of the motivation, the objectives and a brief description of the approach used in the UQOS framework, to account for the effect of the environment on the QoS of software components. A more detailed look at the effect of

the environment on the QoS is presented in chapter 5, which deals with a case-study involving components from the math domain.

4.3. Effect of usage patterns

In this section, the effect of the usage pattern (i.e., the pattern of users and user requests received by components) on the QoS of software components is described. Also presented in this section, are the motivation and the objectives behind the study of the effect of usage patterns, along with the approach used in the UQOS framework to account for the effect of usage patterns on the QoS of software components.

4.3.1. Motivation

Once a component is deployed on the network by the component user, it may be subjected to varying usage patterns. For instance, an e-commerce component such as a credit card verification component, once deployed on the Internet, may be subject to varying number of users and user requests depending on factors like the time of the day, the time of the year (seasonal variation), the deployment site, the semantics of the application etc. The variations in the pattern of users and user requests (the usage patterns) can have a profound impact on the QoS of a component (and in turn, the level of satisfaction of the end-user or consumer). This in effect implies that it is crucial to be able to deduce the effect of usage patterns on the QoS of software components.

Also, a component user interested in improving or maintaining the QoS of a component under different usage patterns, would frequently resort to techniques like investing in more hardware resources. During this activity, any information relating to the behavior of the given component under different usage patterns would prove to be useful to determine the optimal balance between the QoS of the component versus the investment made by the component user in terms of hardware resources. For instance, suppose the component user requires a maximum turn-around-time of ‘t’ milliseconds from a component and it is known that the component offers a turn-around-time of ‘t’

milliseconds for ‘n’ users. Then, the component user could configure the hardware such that, each instance of the component has a maximum of ‘n’ users at any given time, so as to receive a maximum turn-around-time of ‘t’ milliseconds from the component.

Thus, a study of the effect of usage patterns on the QoS of software components can offer significant benefits to the users of the components.

4.3.2. Objectives

In this section, the objectives of the study of the effect of usage patterns on the QoS of software components are presented. These objectives can be stated as follows:

- To create a standard mechanism to account for the effect of usage patterns on the QoS of software components.

- To include the effect of usage patterns on the QoS as an integral part of the UniFrame approach.

- To force a component developer, adhering to the UniFrame approach, to consider the effect of usage patterns on the QoS of his components.

- To provide the component user with a means to deduce the QoS of a given component under specific usage patterns.

- To act as a guide to the component user interested in enhancing or maintaining the QoS of a component, by investing in more hardware resources.

4.3.3. Approach

The UQOS framework prescribes a specific set of steps to be followed by a component developer, adhering to the UniFrame approach, in order to account for the

effect of environment on the QoS of his components. According to the UQOS framework, the component developer has to adopt the following approach for developing a software component with a specific functionality, for a given domain, adhering to the domain model for the chosen domain. This is a continuation of the approach presented in section 4.2.3. The approach related to the effect of usage patterns starts from step 5 as shown below:

5. Effect of usage patterns:

   For each parameter P_i (i=1 to n),

   a. If P_i is dynamic,

      Vary the usage patterns as follows:

      i. Vary the number of users ‘n’ of the component, for a constant rate of requests.
      ii. Run the instrumentation code and record the value of the parameter P_i for each value of ‘n’.
      iii. Plot a graph of P_i versus ‘n’.
      iv. Prepare a table with values of ‘n’ along the rows and the values of P_i along the columns.
      v. Vary the rate of requests ‘r’ for the component, for a constant number of users.
      vi. Run the instrumentation code and record the value of the parameter P_i for each value of ‘r’.
      vii. Plot a graph of P_i versus ‘r’.
      viii. Prepare a table with values of ‘r’ and the values of P_i.
      ix. Vary the deviation‘d’ of the interval between requests for different distributions of the requests‘t’ (where t=Uniform, Gaussian, Poisson distribution), while maintaining a constant number of users ‘n’.
      x. For each ‘t’

- Run the instrumentation code and record the value of the parameter P_i for each value of 'd'.
- Plot a graph of P_i versus 'd'.
- Prepare tables with values of 'd' and the values of P_i.
- Include the prepared tables in the UniFrame description of the component.

The approach presented above satisfies the objectives stated in section 4.2.2. as follows:

1. The approach ensures that the effect of usage patterns on the QoS of software components is considered, by making it an integral part of the UniFrame component development process.
2. By the inclusion of the effect of usage patterns information in the UniFrame description of components, the component developer is forced to account for the effect of usage patterns, in order to comply with the UniFrame approach. The component developer's desire to compete with other component developers complying with the UniFrame guidelines acts as another incentive to persuade him/her to account for the effect of usage patterns on the QoS of his components.
3. A component user interested in deducing the QoS of a component under a given usage pattern, would be able to obtain that information from the table included in the UniFrame description of the component, containing the values of each dynamic QoS parameter against different usage patterns.
4. A component user intending to enhance or maintain the QoS of a component, can identify those usage patterns which would lead to the desired QoS. Then, by re-configuring the hardware resources, the component user can ensure that each instance of the component is subjected to the identified usage pattern, in order to obtain the desired QoS from the component.

This section presented an overview of the motivation, the objectives and a brief description of the approach used in the UQOS framework, to account for the effect of

usage patterns on the QoS of software components. A more detailed look at the effect of usage patterns on QoS of software components is presented as a case-study in chapter 5.

4.4. Specification of QoS of Software Components

Sections 4.1, 4.2 and 4.3 of this chapter dealt with the QoS Catalog, the effect of environment and the effect of usage patterns respectively. In this section, the issue of specification of the QoS of software components is addressed. This section presents the requirements for a QoS specification scheme, followed by the details of the specification scheme adopted.

4.4.1. Requirements

Several interface definition languages (IDLs) (Object Management Group's IDL[CID02], Microsoft IDL[MID02]) allow the representation of the functional aspects of a software component. Along similar lines, there is a need for a formal language to specify the non-functional or QoS aspects of a software component. This is especially true in the context of the UniFrame approach where, QoS is an integral part of every software component. [BEU99] suggests four levels of contracts for software components, namely, basic or syntactic contracts, behavioral contracts, synchronization contracts, and QoS contracts. The notion of service & service guarantees, as used in the UMM, corresponds to the QoS contract proposed in [BEU99].

Any QoS specification language, in the context of the UniFrame approach, has to satisfy the following requirements:

1. It should be generic, i.e., it should not be restricted to particular domains or specific QoS parameters.
2. It should be platform independent, i.e., it should not be tied to specific implementation technologies.
3. It should support the separation of concerns, i.e., the QoS specification must be syntactically and semantically separate from the functional specifications.

4. It should seamlessly integrate with concepts of object-oriented analysis and design.
5. It should be compatible with existing interface definition languages like CORBA IDL.

4.4.2. Specification Scheme

The specification scheme chosen for the UQOS framework is the Component Quality Modeling Language (CQML) [AAG01]. This section is focused on presenting the specifics of the CQML as depicted in [AAG01].

CQML is a lexical language for specifying QoS. It is based on four specification constructs namely, *QoS characteristics*, *QoS statements, QoS profiles* and *QoS categories*. Each of these constructs is explained in detail below.

i. QoS characteristic: A QoS characteristic corresponds to a QoS parameter as used in the context of UQOS. It one of the basic building blocks of a QoS specification. It is a user-defined data type based on one of the fundamental data types namely, *numeric*, *set* or *enum*. These fundamental types specify a domain of values that a QoS characteristic can belong to. A numeric domain is further sub divided into real, integer or natural. Both enum and set have values adopted from a set of user-defined names. The difference between them being that an instance of type enum is one member of the set of user-defined names, while an instance of the set type is one member of the power set of the user-defined names.

   Further, the QoS characteristic can be defined as either *increasing* or *decreasing*. A QoS characteristic is defined as increasing, if higher values correspond to better QoS and vice versa. A QoS characteristic is defined as decreasing, if lower values correspond to higher QoS and vice versa.

   The grammar for a QoS characteristic as illustrated in [AAG01] is shown in figure 4.2.:

```
<characteristic_declaration>   ::= quality_characteristic <characteristic_name>
                                   '{'<characteristic_body> '}'
<characteristic_body>          ::= <domain_declaration>
<domain_declaration>           ::= domain ':' <domain_type> ';'
<domain_type>                  ::= numeric | set | enum
```

Figure 4.2. Grammar for CQML QoS characteristic

The grammar for the numeric domain as illustrated in [AAG01] is:

```
<domain_type>          ::= <numeric> | set | enum
<numeric>              ::= numeric [<type_specification>] [<boundary_restriction>]
<type_specification>   ::= real
                       |   integer
                       |   natural
<boundary_restriction> ::= <lower_boundary> <OCL::number> '..' [ <OCL::number> ]
                           <upper_boundary>
                       |   <lower_boundary> '..' <OCL::number> <upper_boundary>
```

Figure 4.3. Grammar for the numeric domain

The grammar for the numeric domain is further extended in [AAG01] as shown in figure 4.4.

```
<numeric>      ::= [<direction>] numeric [<type_specification>]
                   [<boundary_restriction>]
<direction>    ::= increasing | decreasing
```

Figure 4.4. Extended grammar for the numeric domain

A sample QoS characteristic, Turn-around-time can be defined in CQML as follows:

```
quality_characteristic Turn-around-time {
        domain: decreasing numeric real;
}
```

Here, it can be seen that the QoS parameter Turn-around-time has been defined as a QoS characteristic with values from the domain of real numbers, with lower values signifying higher QoS and vice versa.

CQML uses the Object Constraint Language (OCL) as defined in [UML99] to specify invariants which are properties inherent to a QoS

characteristic, valid for any measurement. For example the Turn-around-time could be constrained to be always less than 100 milliseconds by the OCL statement:

self <= 100

where, self refers to Turn-around-time.

ii. QoS statement: QoS statements are used to specify constraints on the QoS characteristics in order to represent the total QoS of a component. The grammar for the QoS statement as illustrated in [AAG01] is shown in figure 4.5.

```
<quality_declaration>  ::=  quality <quality_name> '{' {<constraint> ';' }* '}'
<constraint>           ::=  <simple_constraint> { <OCL::logicalOperator> <constraint> }*
                       |    '(' <constraint> ')'
<simple_constraint>    ::=  <characteristic_name> <OCL::relationalOperator> <value>
<value>                ::=  <element> | <range_values> | <characteristic_name>
```

Figure 4.5. Grammar for CQML QoS statement

CQML provides the option of including a unit (like milliseconds) for the values of a domain. However, these units are not predefined and they are represented in the grammar as non-terminals. A compiler and run-time system for conformance checking can be extended to reason about the units.

A sample QoS statement can be defined in CQML as follows:

```
quality low_Turn-around-time {
        Turn-around-time <=10 milliseconds;
}
```

It can be seen in the QoS statement above that the QoS parameter Turn-around-time has been constrained to be less than 10 milliseconds.

iii. QoS profile: QoS profiles are used to associate QoS statements with component specifications. The grammar for a QoS profile as illustrated in [AAG01] is shown in figure 4.6.

```
<profile>              ::= profile <profile_name> for <component_name>
                           '{' <profile_body> '}'
<profile_body>         ::= <offer_specification>
<offer_specification>  ::= provides <offer> ';'
<offer>                ::= <offer_name> [ '(' <OCL::actualParameter> ')' ]
                           {<OCL::logicalOperator> <offer> }*
                           | '(' <offer> ')'
```

Figure 4.6. Grammar for CQML QoS profile

Let the CORBA IDL interface for a matrix addition component be defined as follows:

```
interface MatrixAddition {
        float[ ][ ]  matrixAddition(in float[ ][ ]  matrix1, in float[ ][ ] matrix2);
};
```

A component which implements the above interface can be defined using the Component Interface Definition Language (CIDL) [OMG99] as follows:

```
Component myMatrixAddition {
        Provides MatrixAddition matAdd;
};
```

Let a QoS statement for myMatrixAddition be defined as:

```
quality min_Turn-around-time {
        Turn-around-time < 15 milliseconds;
}
```

Now, a QoS profile for the component myMatrixAddition can defined as follows:

```
profile goodMatrixAddition for myMatrixAddition {
        provides min_Turn-around-time (matAdd);
}
```

The above profile states that the component myMatrixAddition can provide a min_Turn-around-time, i.e., a Turn-around-time of less than 15 milliseconds.

iv. QoS category: These are used to group together QoS characteristics, QoS statements and QoS profiles belonging to a certain domain or satisfying certain properties under named categories. This is analogous to the classification structure used in the QoS Catalog, based on the domain of usage, the static or dynamic behavior, the nature of the parameters (according to [OMG02]) and Composability of the parameters. The grammar for the QoS category as illustrated in [AAG01] is shown in figure 4.7.

```
<category_declaration>  ::=  quality_category <category_name>
                             '{' <category_body> '}'
<category_body>         ::=  <cqml_declaration>+
                        |    <cqml_reference>+
<cqml_reference>        ::=  <characteristic_name> ';'
                        |    <quality_name> ';'
                        |    <profile_name> ';'
                        |    <category_name> ';'
```

Figure 4.7. Grammar for CQML QoS category

The grammar for <cqml_declaration> as illustrated in [AAG01] is given in figure 4.8.

```
<cqml_declaration>      ::=  <category_declaration>
                        |    <characteristic_declaration>
                        |    <quality_declaration>
                        |    <profile_declaration>
```

Figure 4.8. Grammar for <cqml_declaration>

For instance, the QoS characteristics Turn-around-time and Throughput can be classified under the performance category as follows:

```
quality_category performance {
        Turn-around-time;
        Throughput;
}
```

CQML satisfies the requirements stated in section 4.4.1. as follows:

1. It is generic, i.e., it can be used to describe any QoS parameter and is not restricted to any specific domain.

2. It is platform independent, i.e., it does not rely on any features specific to an implementation technology.
3. It supports separation of concerns, i.e., the QoS specification of a component is separate from the functional specification of the component.
4. The CQML constructs QoS characteristic, QoS statement and QoS profile all support object oriented concepts like encapsulation and specialization.
5. It is compatible with existing interface definition languages like CORBA IDL and Microsoft IDL.

This chapter presented a description of the implementation of the UQOS framework, consisting of four parts, namely, the QoS Catalog for software components, the approach for accounting for the effects of environment on the QoS of software components, the approach for accounting for the effects of usage patterns on the QoS of software components and the specification of the QoS of software components. In the next chapter (Chapter 5), a case-study from the math domain is provided to illustrate the applicability of the UQOS framework in a real-world scenario and to link the CQML to the UniFrame approach.

5. CASE-STUDY

Chapters 3 and 4 dealt with the UniFrame approach and the implementation of the UQOS framework respectively. This chapter is intended to provide a case-study to illustrate the application of UniFrame approach and the UQOS framework in a real-world scenario.

The case-study uses the Simics simulator [SIM02] to perform the effect of environment related experiments. Simics is a full-system simulator that is capable of simulating various machine configurations, allowing the user to control the processor speed, memory and other system configurations. It can also run unmodified operating systems such as Solaris, Linux and Windows on the simulated machine (called the target machine). In the case-study, the Simics is operating on an Intel Pentium 4, 1.6 GHZ processor machine (called the source machine) running Windows XP. The target machine being simulated is an Intel Pentium 2 with various processor speeds and memory. The operating system running on the target machine is the Red Hat Linux version 7.1.

The experiments were carried out as follows:

1. Set the processor speed to the required clock frequency and the memory to the required megabytes in the Simics configuration file (acpi-machine-generic.simics). The process priority is set using the Linux command '*nice'*.
   Using the nice command, the priority of a process can be set between -20 to 19, with -20 being the highest priority and 19 the lowest priority.
2. Boot the target system, the boot-up time varies from approximately 30 minutes up to 4 hours depending on the speed of the processor being simulated (higher the speed, the longer the boot-up time) and the configuration of the source machine.
3. Run the components and the instrumentation code. Record the values of the QoS parameters for the corresponding values of the environment variables.

4. Plot the graphs of the values of the QoS parameters against the environment variables.
5. Create the tables of the values of the QoS parameters against the environment variables.
6. Repeat steps 1 to 5 for different values of environment variables.

The experiments related to the effect of usage patterns were conducted using the Apache JMeter tool [JME02]. The JMeter is a performance measurement tool that can simulate various users and user request patterns on a software component. It also provides the user with performance characteristics like the latency and the throughput of the component. The case-study uses the JMeter version 1.8, running on an Intel Pentium 4, 1.6 GHZ processor machine running Windows XP. The components are running on the Apache Tomcat Servlet Engine [TOM02]. The Tomcat is running on the same host as the JMeter. JMeter's definitions of the latency and throughput, as indicated in [JME02], are similar to the definitions of Turn-around-time and Throughput in the QoS Catalog. Hence, the JMeter is used as the tool to study the effect of usage patterns, in this case-study.

The experiments were carried out as follows:

1. Start the Tomcat servlet engine.
2. Start the JMeter.
3. Configure the JMeter to run the required components.
4. Set the number of users and the pattern of user requests on the JMeter.
5. Run the components using the JMeter and record the values of the QoS parameters.
6. Plot the graphs of the values of the QoS parameters against the usage patterns.
7. Create the tables of the values of the QoS parameters against the usage patterns.
8. Repeat steps 4 to 7 for different usage patterns.

The component considered here, is from the math domain, involved with matrix operation, namely, matrix multiplication. The approach used by a component developer following the UniFrame approach, to develop these components is as follows:

1. The component developer would start out by referring to the domain model from the math domain for the matrix operations, in particular the matrix multiplication operation.

2. The domain model would provide the component developer with the standardized specifications that are to be incorporated into the component interfaces.

3. The CORBA IDL descriptions of the interfaces of the matrix multiplication component is as follows:

```
interface MatrixMultiplication {
        double[ ][ ] matrixMultiply(in double[ ][ ] matrix1, in double[ ][ ]
        matrix2);
};
```

4. The component developer would now provide the implementation for the MatrixMultiplication interface using the programming language of his choice. In this case-study, the implementation language chosen is Java. The IDL to Java compiler *'idlj'* can be used to map the IDL interfaces to Java. The idlj compiler is included with the Java Standard Development Kit (SDK1.4) by Sun Microsystems and it is aligned with CORBA version 2.3.2. The idlj compiler produces several files as output including the server skeleton, client stub and other CORBA- to-Java mapping classes. For the sake of simplicity, just the code corresponding to the matrix multiplication server component is presented. This code should be used in conjunction with the code produced by the idlj compiler.

   The code for the matrix multiplication server component is as follows:

```
// The package containing stubs.
import MatrixMultiplicationApp.*;
// MatrixMultiplicationServer will use the naming service.
import org.omg.CosNaming.*;
// The package containing special exceptions thrown by the name service.
import org.omg.CosNaming.NamingContextPackage.*;
// All CORBA applications need these classes.
import org.omg.CORBA.*;

public class MatrixMultiplicationServer
{
 public static void main(String args[ ])
 {
  try{

    // Create and initialize the ORB
    ORB orb = ORB.init(args, null);

    // Create the servant and register it with the ORB
    MatrixMultiplicationServant MatrixMultiplicationRef = new
        MatrixMultiplicationServant();
    orb.connect(MatrixMultiplicationRef);

    // Get the root naming context
    org.omg.CORBA.Object objRef = orb.resolve_initial_references ("Name
        Service");
    NamingContext ncRef = NamingContextHelper.narrow(objRef);

    // Bind the object reference in naming
    NameComponent nc = new NameComponent("MatrixMultiplication", "");
```

```
    NameComponent path[] = {nc};
    ncRef.rebind(path, MatrixMultiplicationRef);

    // Wait for invocations from clients
    java.lang.Object sync = new java.lang.Object();
    synchronized(sync){
      sync.wait();
    }

   } catch(Exception e) {
      System.err.println("ERROR: " + e);
      e.printStackTrace(System.out);
    }
  }
}

class MatrixMultiplicationServant extends _MatrixMultiplicationImplBase
{
  public double[ ][ ] matrixMultiply(double matrix1[ ][ ], double matrix2[ ][ ])
  {
          int size = matrix1.length;
          double product[ ][ ] = new product[size][size];
          mmul mymmul=new mmul;
          TurnAroundTime tat=new TurnAroundTime();
          double t1=System.currentTimeMillis();
          sum= mymmul.MultiplyMatrix(matrix1,matrix2);
          double t2=System.currentTimeMillis();
          //call Instrumentation code
          double t3= tat.findTurnAroundTime(t1,t2);
          System.out.println("Turn-around-time in milliseconds:" +t3);
```

```
        //call Instrumentation code
        Throughput tp=new Throughput();
        System.out.println("Throughput in results/second:" +
        tp.findThroughput(t3));
        return (product);

  }
}

class mmul {

public double[ ][ ] MultiplyMatrix(double m1[ ][ ], double m2[ ][ ]){

        if (m1.length != m2.length) {return; }
        int size=m1.length;
        double result[ ][ ] = new double[size][size];
        for (int i=0; i < size; i++){
                for (int j=0; j < size; j++) {
                        result[i][j]=0.0;
                }
           }
           for (int i=0; i < size; i++){
                for (int j=0; j < size; j++){
                        for (int p=0; p < size; p++){
                                result[i][j] += m1[i][p] * m2[p][j];
                        }
                }
           }
```

```
            return(result);
        }
    }
```

5. The component developer would now refer to the QoS Catalog and identify the QoS parameters of relevance to the domain under consideration. For this case-study, let us assume that the parameters chosen are: Turn-around-time and Throughput.

6. The component developer would now create/acquire the QoS instrumentation code for each of the chosen parameters, adopting the QoS quantification models prescribed in the QoS Catalog. The components would pass the required parameters to the instrumentation code and the instrumentation code would return the relevant QoS values. The instrumentation code for Turn-around-time and Throughput are as follows:

   Instrumentation code for Turn-around-time:

```
package qoscat;

public class TurnAroundTime {

public double findTurnAroundTime(double t1, double t2){
        double t3;
        t3=t2-t1;
        return (t3);
  }
}
```

Instrumentation code for Throughput:

```
package qoscat;
import java.util.ArrayList;

public class Throughput {

public double findThroughput(double t) {
        double tp;
        tp=(1000/t);
        return (tp);
          }
 }
```

7. The component developer would now run the components and the instrumentation code and use the approach presented in sections 4.2 and 4.3 to account for the effect of the environment and the effect of usage patterns on these components.

   The graphs obtained as a result of applying the above-mentioned approach to this case-study (for two matrices of size 10x10) are presented below in the following order:

   For the effect of environment on the Turn-around-time:

   1. Effect of CPU speed on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.1.).
   2. Effect of memory on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.2.).
   3. Effect of CPU speed and memory together on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.3.).
   4. Effect of process priority on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.4.).

For the effect of usage patterns on the Turn-around-time:

1. Effect of number of users on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.5.).
2. Effect of delay between requests on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.6.).
3. Effect of maximum delay between requests with Uniform distribution, on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.7.).
4. Effect of Maximum delay between requests with Gaussian distribution, on the Turn-around-time of the MatrixMultiplicationServer component (figure 5.8.).

For effect of the environment on the Throughput:

1. Effect of CPU speed on the Throughput of the MatrixMultiplicationServer component (figure 5.9.).
2. Effect of memory on the Throughput of the MatrixMultiplicationServer component (figure 5.10.).
3. Effect of CPU speed and memory together on the Throughput of the MatrixMultiplicationServer component (figure 5.11.).
4. Effect of process priority on the Throughput of the MatrixMultiplicationServer component (figure 5.12.).

For the effect of usage patterns on the Throughput:

1. Effect of number of users on the Throughput of the MatrixMultiplicationServer component (figure 5.13.).
2. Effect of delay between requests on the Throughput of the MatrixMultiplicationServer component (figure 5.14.).
3. Effect of maximum delay between requests with Uniform distribution, on the Throughput of the MatrixMultiplicationServer component (figure 5.15.).
4. Effect of maximum delay between requests with Gaussian distribution, on the Throughput of the MatrixMultiplicationServer component (figure 5.16.).

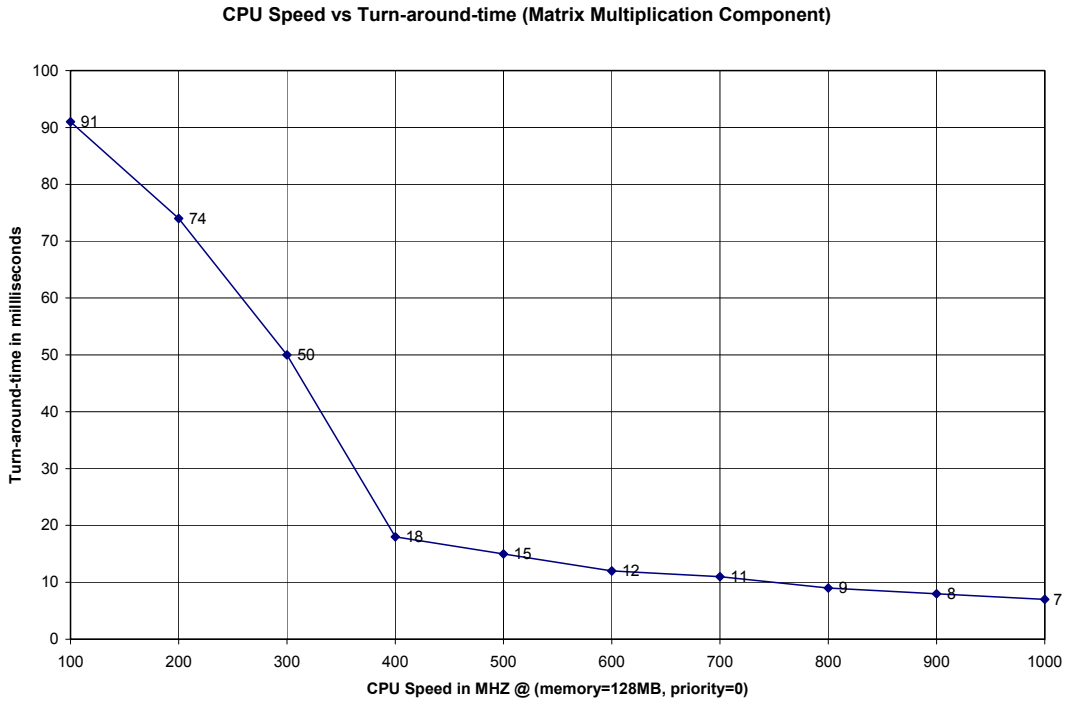

Figure 5.1. CPU Speed vs. Turn-around-time

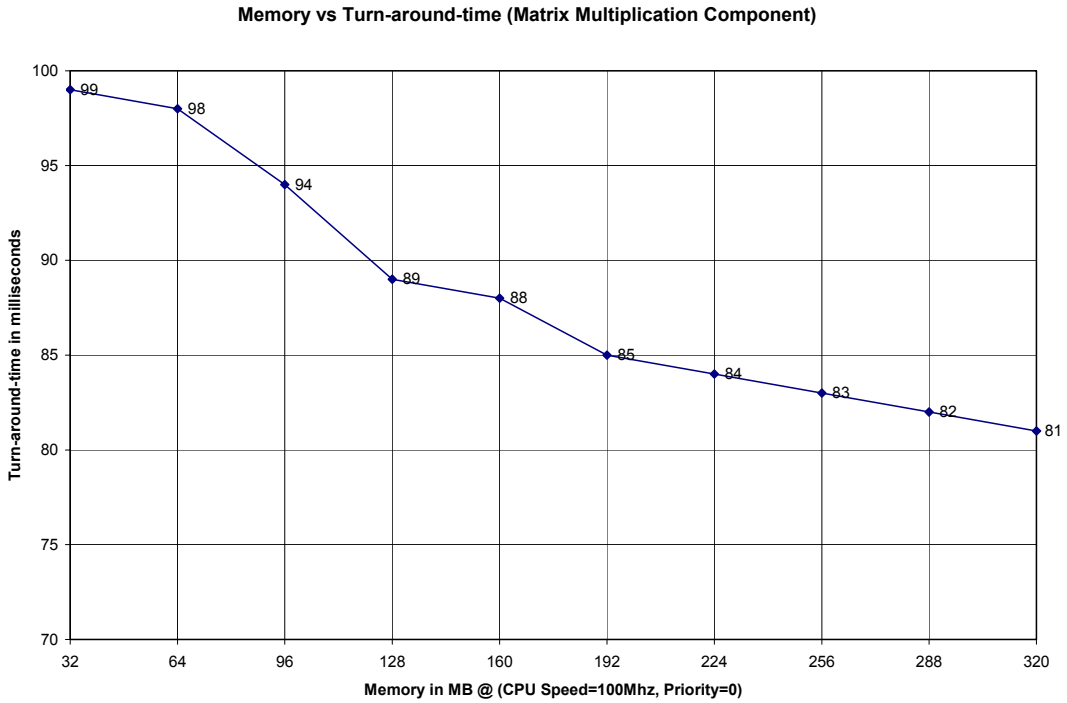

Figure 5.2. Memory vs. Turn-around-time

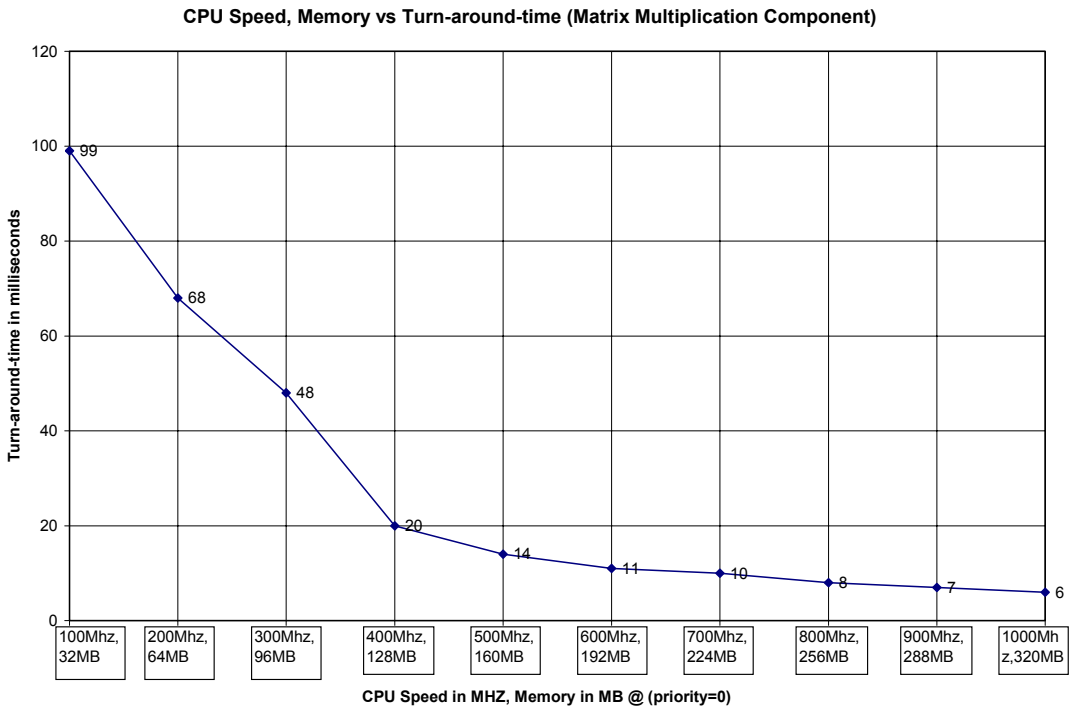

Figure 5.3. CPU speed, Memory vs. Turn-around-time

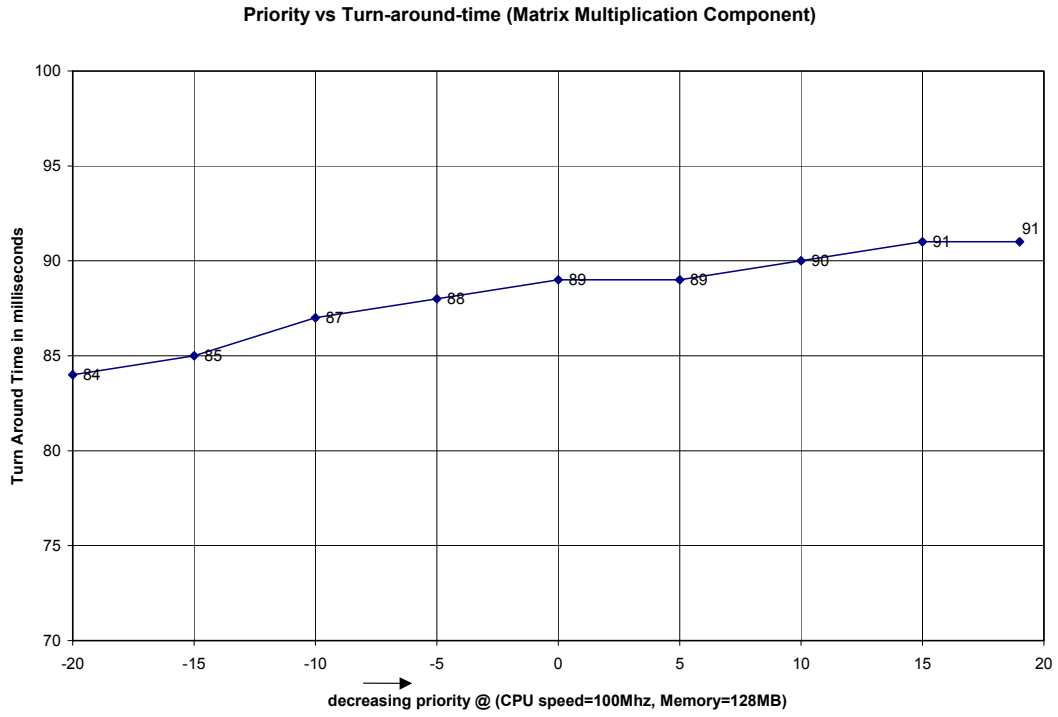

Figure 5.4. Priority vs. Turn-around-time

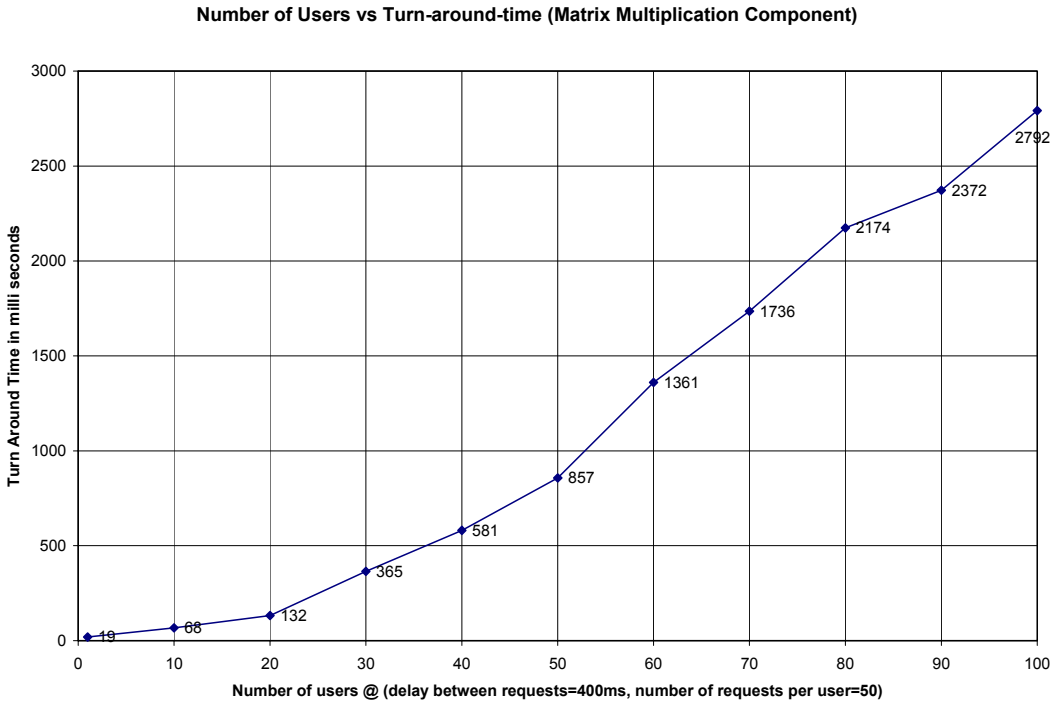

Figure 5.5. Number of users vs. Turn-around-time

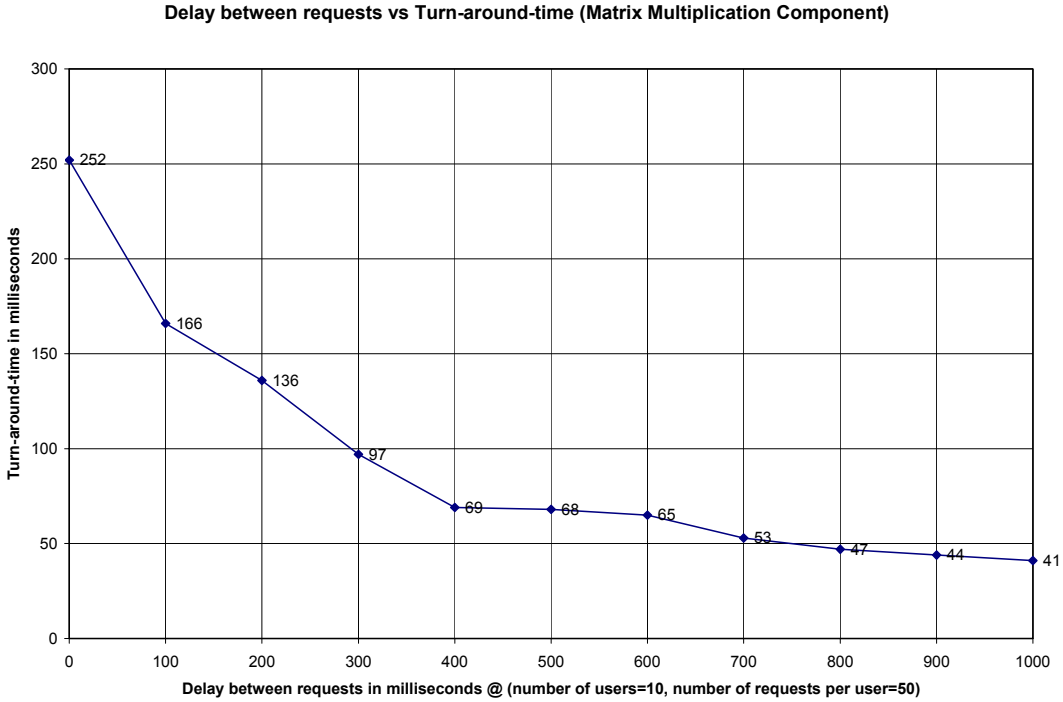

Figure 5.6. Delay between requests vs. Turn-around-time

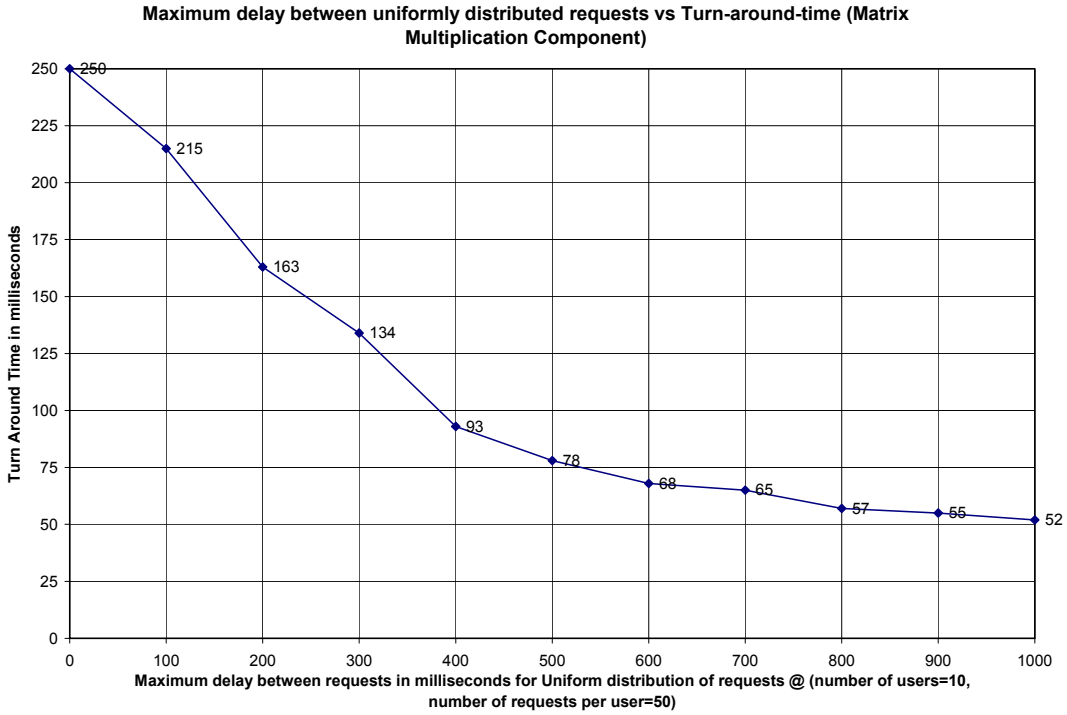

Figure 5.7. Maximum delay between uniformly distributed requests vs. Turn-around-time

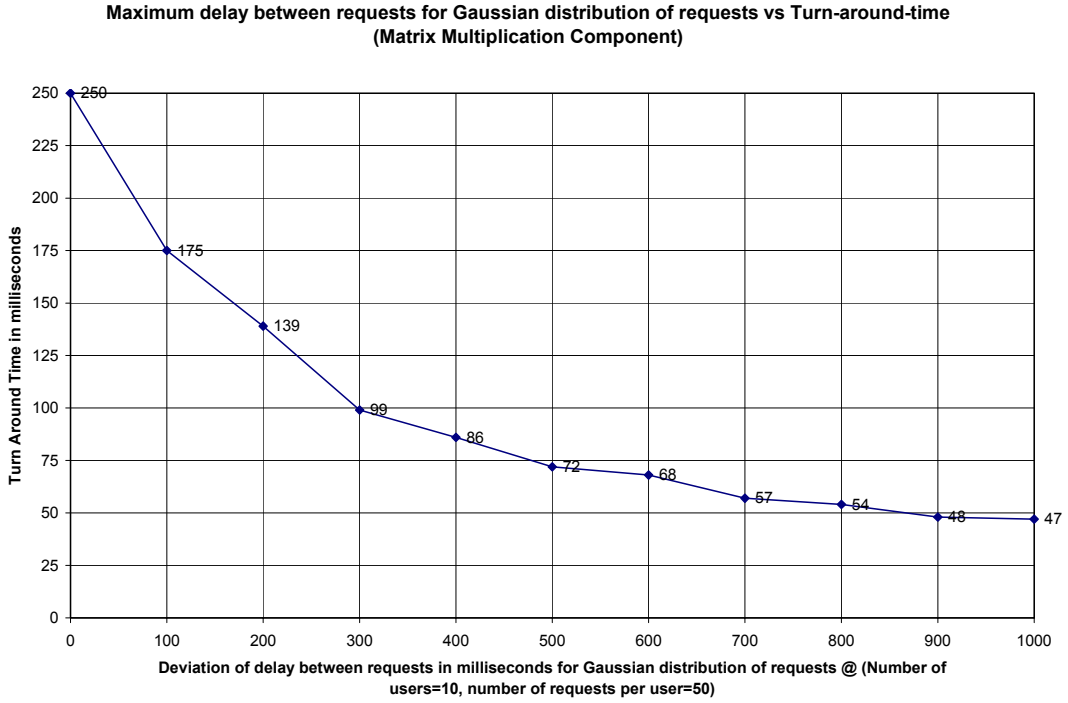

Figure 5.8. Maximum delay between requests for Gaussian distribution of requests vs. Turn-around-time

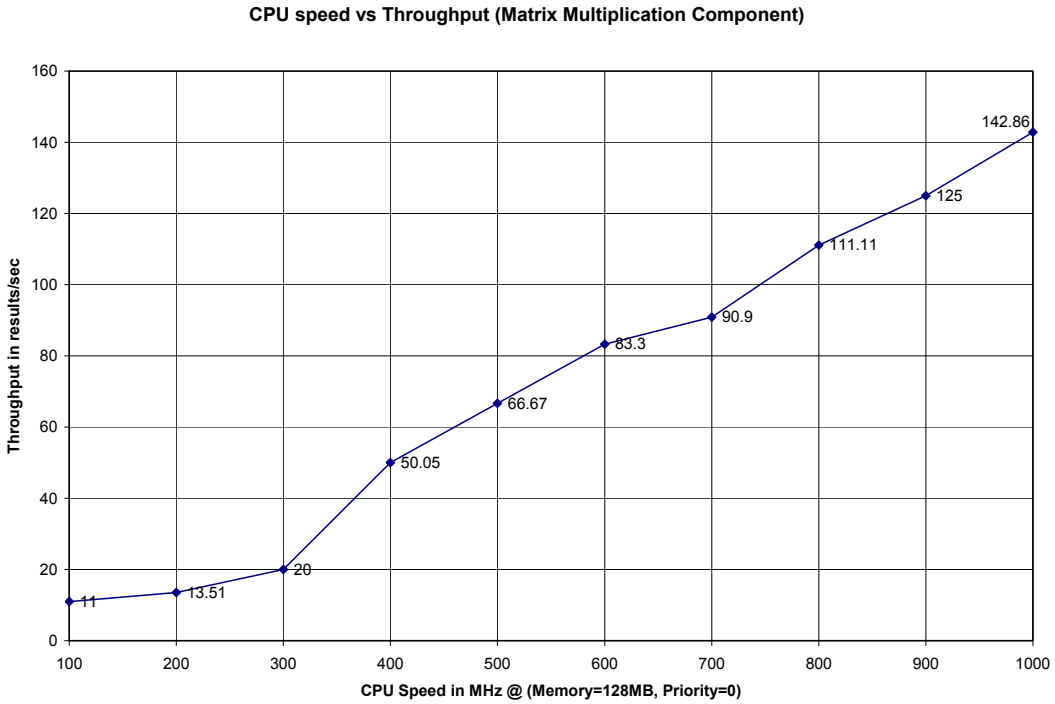

Figure 5.9. CPU speed vs. Throughput

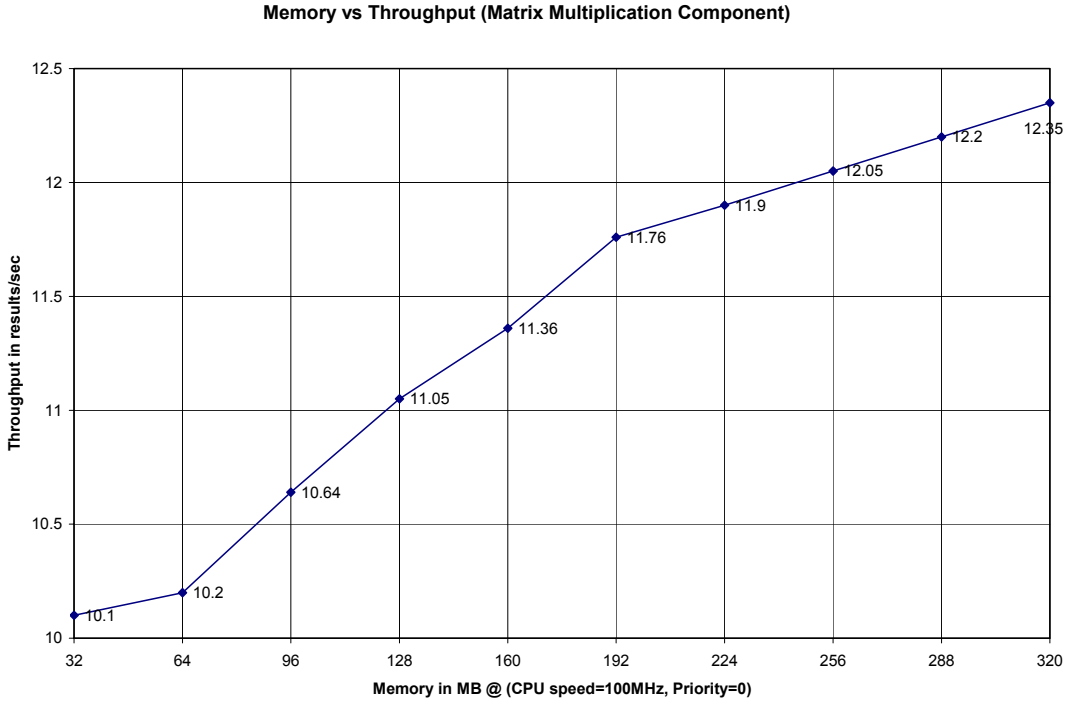

Figure 5.10. Memory vs. Throughput

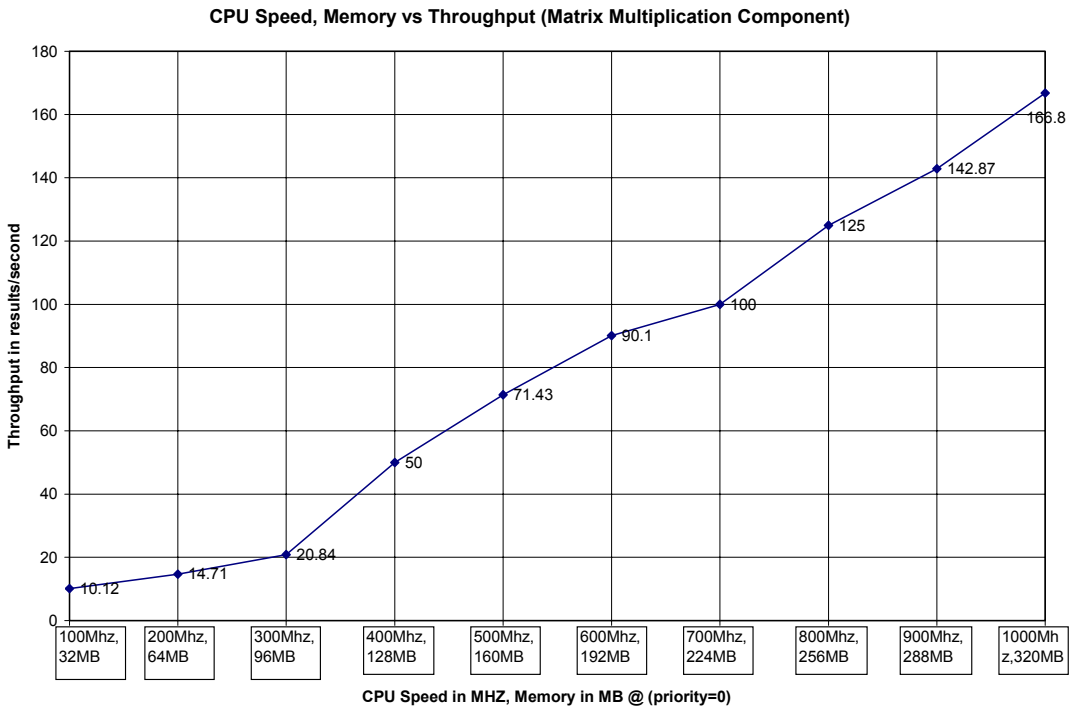

Figure 5.11. CPU speed and Memory vs. Throughput

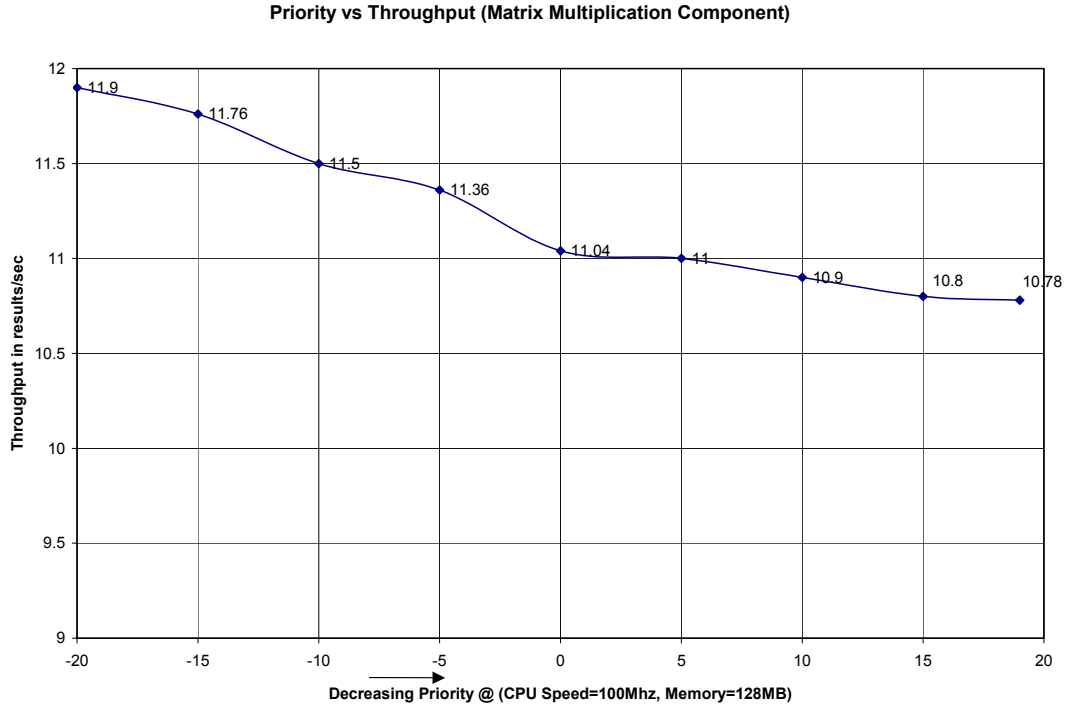

Figure 5.12. Priority vs. Throughput

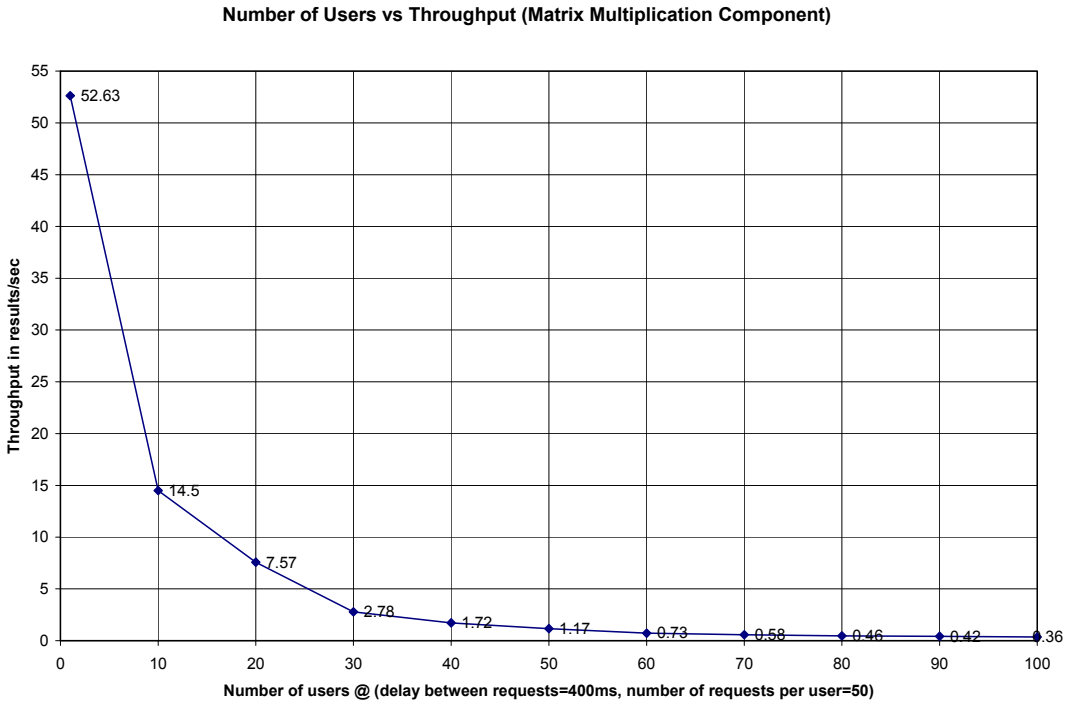

Figure 5.13. Number of users vs. Throughput

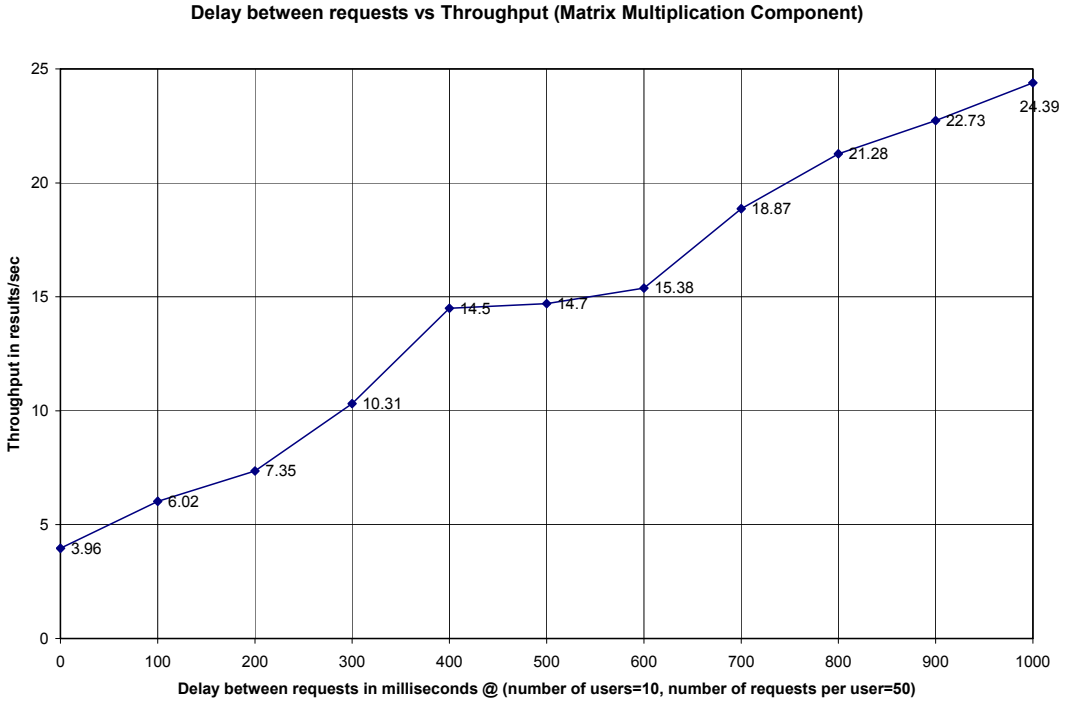

Figure 5.14. Delay between requests vs. Throughput

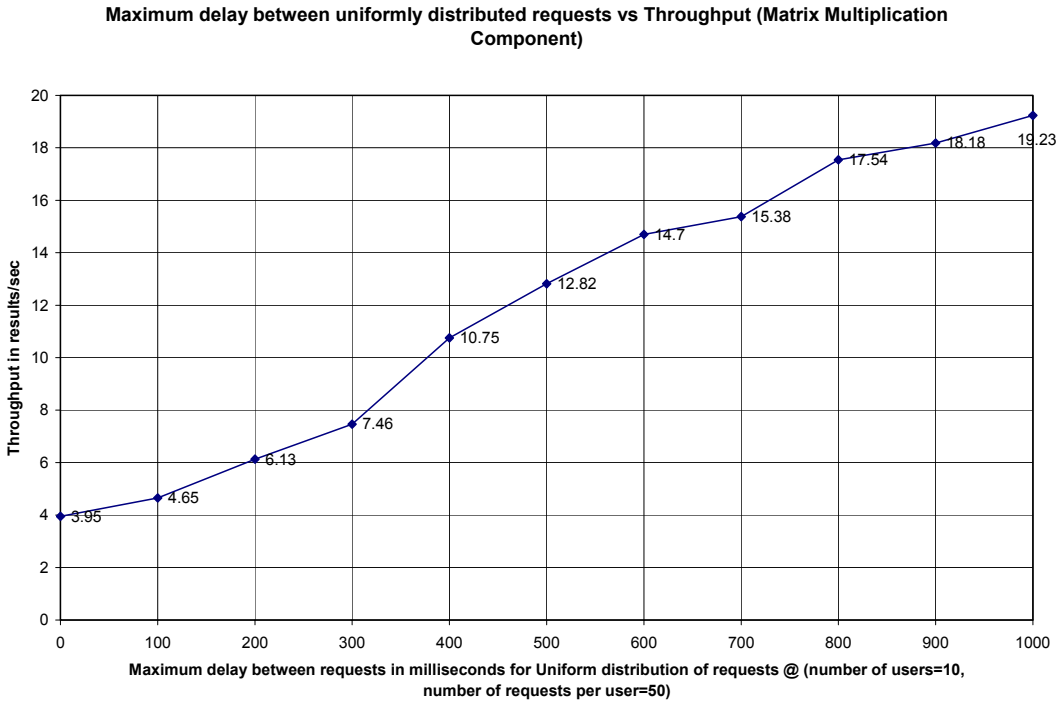

Figure 5.15. Maximum delay between requests for uniformly distributed requests vs. Throughput

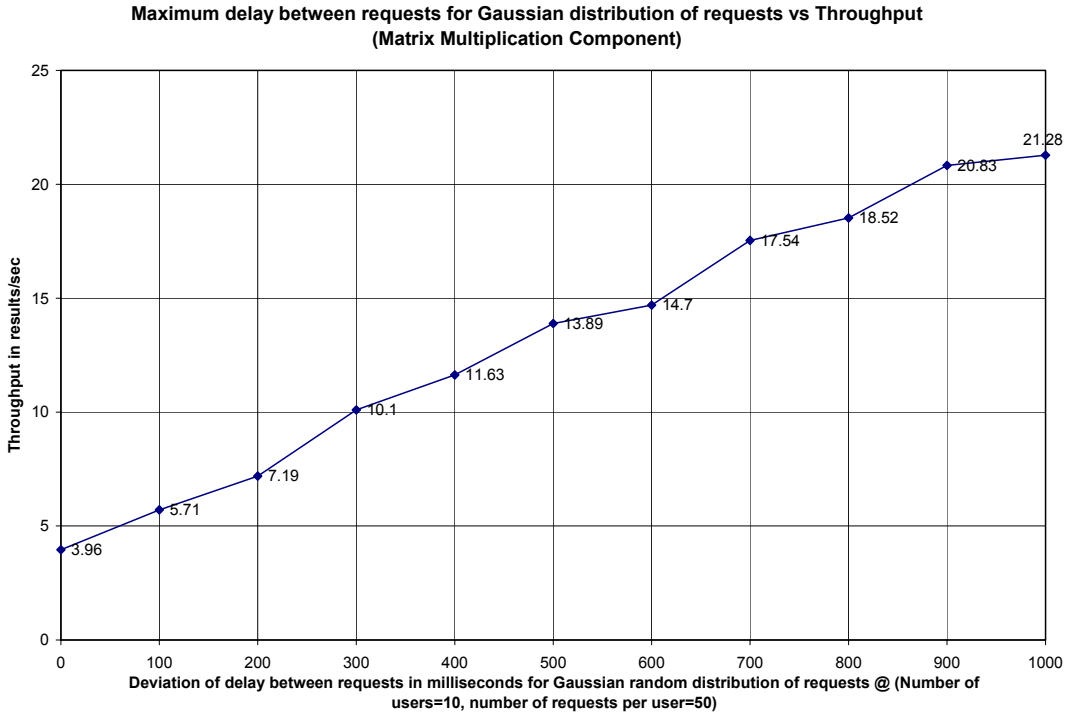

Figure 5.16. Maximum Delay between requests for Gaussian distribution of requests vs. Throughput

7.1. Analysis of graphs

7.1.1. CPU speed vs. Turn-around-time (figure 5.1.):

As seen in the graph, the Turn-around-time of the matrix multiplication component decreases with increase in the CPU speed. This is in line with the general belief that components execute faster on faster processors, leading to lower Turn-around-time. This is attributed to the fact that, a processor with a higher clock rate can process more machine-level instructions per unit time than a processor with a lower clock rate.

7.1.2. Memory vs. Turn-around-time (Figure 5.2.):

The graph shows a decrease in the Turn-around-time of the component as the memory increases. This is attributed to a decrease in the frequency of data-swapping between the hard-disk and the memory, with increase in memory size. The disk-access-time can be a significant part of the Turn-around-time, but with higher memory, the disk access is reduced, leading to lower Turn-around-time. It can be further seen that the values of Turn-around-time for (100MHZ, 128MB, 0 priority) in the Figures 5.1 and 5.2 correspond to each-other.

7.1.3. CPU speed and memory vs. Turn-around-time (Figure 5.3.):

It can be seen from the graph that, there is a decrease in Turn-around-time, with a simultaneous increase in CPU speed and memory. This can be deduced from the graphs in Figures 5.1 and 5.2, both of which show a decrease in Turn-around-time with increase in CPU speed and Memory respectively. It can be further seen that the values of Turn-around-time for (400MHZ, 128MB, 0 priority) in the Figures 5.1, 5.3 and for (100MHZ, 32MB, 0 priority) in the Figures 5.2 and 5.3 correspond to each-other.

7.1.4. Priority vs. Turn-around-time (Figure 5.4.):

The graph shows an increase in Turn-around-time with decreasing priority. Every process executing on a machine has a priority assigned to it (by the user or the

operating system). The priority value is a reflection of the preference given by the operating system to a given process over other processes with lower priority. Processes with higher priority get more resources and are served faster (less waiting in queues). Hence, higher the priority assigned to the component, the lower the Turn-around-time and vice-versa. It can be further seen that, values of Turn-around-time for (100MHZ, 128MB, 0 priority) in the Figures 5.1, 5.2 and 5.4 correspond to each-other.

7.1.5. Number of users vs. Turn-around-time (Figure 5.5.):

It can be seen in the graph that there is a steady increase in the Turn-around-time of the matrix multiplication component with increase in the number of users. A multi-threaded component handles multiple requests at a time by creating separate threads to handle each of the user requests. But the number of threads that can be created is limited by the resources of the host machine on which the component is deployed. Hence, when a component receives more requests than it can handle (due to resource limitation), it usually inserts them in a queue. Hence, higher the number of user requests, the higher the likelihood of the request being queued, resulting in higher Turn-around-time.

7.1.6. Delay between requests vs. Turn-around-time (Figure 5.6.):

The graph shows a steady decrease in the Turn-around-time of the component with increase in the delay between requests. This can be attributed to the fact that, the longer the delay between the requests, the lower the chances of the component not being able to serve the request, and the request ending up in a queue. Hence, the decrease in Turn-around-time with increase in the delay between requests. It can be further seen that, values of Turn-around-time for (10 users, 400ms delay between requests, 50 requests per user) in the Figures 5.5 and 5.6 correspond to each-other.

7.1.7. Maximum delay between uniformly distributed requests vs. Turn-around-time (Figure 5.7.):

This graph is intended to show the Turn-around-time of the component for a realistic request pattern. Here, the requests received are according to a uniform random distribution. The x-axis indicates the maximum possible delay between requests for each trial and the actual delay between the requests can be any value between zero to the maximum value. It can be seen that the Turn-around-time values in this graph are greater than or equal to the corresponding values in the Figure 5.6. This is because the Figure 5.6 specifies the fixed delay between requests (from 0 to 1000) whereas this figure specifies the maximum limit for a uniform random distribution of delay between requests (from 0 to 1000).

7.1.8. Maximum delay between requests following a Gaussian distribution vs. Turn-around-time (Figure 5.8.):

This graph shows the variation in Turn-around-time of the component for a realistic request pattern following a Gaussian random distribution. As in Figure 5.7, it can be seen in this figure that the Turn-around-time values are greater than or equal to the corresponding values in Figure 5.6. The reason is, that the Figure 5.6 specifies the fixed delay between requests (from 0 to 1000) whereas this figure specifies the maximum limit for a Gaussian random distribution of delay between requests (from 0 to 1000).

7.1.9. CPU speed vs. Throughput (Figure 5.9.):

This graph shows a steady increase in the throughput of the component with increase in the CPU speed. This can be attributed to the fact that, as the CPU speed increases, the number of instructions that the CPU can process per unit time also increases. Throughput is defined as the number of results returned by the component per second (unit time). The CPU can process more of component instructions per unit time with increase in the clock speed, leading to higher throughput from the component.

7.1.10. Memory vs. Throughput (Figure 5.10.):

It can be seen in the graph that there is an increase in the Throughput of the component with increase in the memory size. With the increase in the memory, the number of disk accesses is reduced and this reduces the overhead associated with the disk access. This in-turn means that the component can access its data much faster, leading to an increase in the number of speed of execution of the component, resulting in higher throughput. It can be further seen that the values of Throughput for (100MHZ, 128MB, 0 priority) in the Figures 5.9 and 5.10 correspond to each-other.

7.1.11. CPU speed and Memory vs. Throughput (Figure 5.11):

The graph shows an increase in Throughput with a simultaneous increase in CPU speed and memory. This result can be directly deduced from the graphs in Figures 5.9 and 5.10, both of which show an increase in Throughput with increase in CPU speed and Memory respectively. It can be further seen that the values of Throughput for (400MHZ, 128MB, 0 priority) in the Figures 5.9, 5.11 and for (100MHZ, 32MB, 0 priority) in the Figures 5.10 and 5.11 correspond to each-other.

7.1.12. Priority vs. Throughput (Figure 5.12):

It is seen from the graph that there is a decrease in the Throughput of the component with decrease in the priority. As the priority of the component decreases, it is more likely that the component would have to wait for resources assigned to processes with higher priority. This results in a decrease in the number of operations the component can perform in a given unit of time and hence a decrease in the number of results the component can produce per second (Throughput). It can be further seen that, the values of Throughput for (100MHZ, 128MB, 0 priority) in the Figures 5.9, 5.10 and 5.12 correspond to each-other.

7.1.13. Number of users vs. Throughput (Figure 5.13.):

The graph shows a decrease in the Throughput of the component with increase in the number of users. As said in the analysis of Figure 5.5, a separate thread is created to handle each request to the component. But, as the number of users (and number of requests) increase, there is an explosion in the number of threads created to handle the requests. This results in the component spending more time spooling the threads, managing the threads and the request queues. This means that the component is spending less time to process the requests. Hence, the number of results the component produces per unit time decreases, resulting in lower throughput.

7.1.14. Delay between requests vs. Throughput (Figure 5.14.):

This graph shows an increase in the throughput of the component with increase in the delay between requests. When the delay between requests is set very low, the component tends to create more threads to handle the requests. As explained in the analysis of Figure 5.13, higher the number of threads, the higher the overhead of thread creation, thread management and queue management. This means that the component spends less time on handling the requests, resulting in low throughput. But, as the delay between requests increases, the number of new threads needed to handle those requests in reduced (because some of the older threads in the thread pool, which have completed their task, may be reused to handle the new requests). This means that the component spends more time in possessing the requests, resulting in an increase in the throughput of the component. It can be further seen that, values of Throughput for (10 users, 400ms delay between requests, 50 requests per user) in the Figures 5.13 and 5.14 correspond to each-other.

7.1.15. Maximum delay between uniformly distributed requests vs. Throughput (Figure 5.15):

This graph is intended to represent the variation in the Throughput of the component for a realistic distribution of requests. Here, the requests received are according to a uniform random distribution. The x-axis indicates the maximum possible delay between requests for each trial. The actual delay between the requests can be any value between zero to the maximum value. It can be seen that the Throughput values in this graph are less than or equal to the corresponding values in the Figure 5.15. This is because the Figure 5.15 specifies the fixed delay between requests (from 0 to 1000) whereas this figure specifies the maximum limit for a uniform random distribution of delay between requests (from 0 to 1000).

7.1.16. Maximum delay between requests following a Gaussian distribution vs. Throughput (Figure 5.8.):

This graph shows the variation in the Throughput of the component for a realistic request pattern following a Gaussian random distribution. As in Figure 5.15, it can be seen in this figure that the Throughput values are greater than or equal to the corresponding values in Figure 5.14. The reason being that the Figure 5.14 specifies the fixed delay between requests (from 0 to 1000) whereas this figure specifies the maximum limit for a Gaussian random distribution of delay between requests (with a mean of 0.0 and a standard distribution of 1.0).

8. The QoS of this component can now be specified using CQML as follows:

```
MatrixMultiplicationServer Component:
quality_characteristic Turn-around-time {
        domain: decreasing numeric real;
}
quality_characteristic Throughput {
```

```
        domain: increasing numeric real;
    }
    quality min_Turn-around-time {
        Turn-around-time <=16 milliseconds;
    }
    quality high_Throughput {
        Throughput >= 60 results/second
    }

    profile goodMatrixMultiplication for myMatrixMultiplication{
        Provides min_Turn-around-time (MatrixMultiplicationServer);
        Provides high_Throughput (MatrixMultiplicationServer);
    }
```

9. The component developer would now create the UMM description for the component and include the information related to the QoS, the effect of the environment and the effect of usage patterns on the QoS of the components, in the UniFrame description.

   The UMM description of the MatrixMultiplicationServer component is shown below:

UMM description for MatrixMultiplicationServer:

1. Name: MatrixMultiplicationServer
2. Domain: Math
3. Informal Description: A matrix multiplication component that provides as output, the product of two input matrices.
4. Computational Attributes:
   - 4.1. Inherent Attributes:
     - 4.1.1. Id: http://magellan.cs.iupui.edu:8080/MatrixMultiplicationServer
     - 4.1.2. Version: 1.0
     - 4.1.3. Author: xyz tech

4.1.4. Date: 10/2/2002

4.1.5. Validity: 10/2/2003

4.1.6. Atomicity: yes

4.1.7. Registration: http://phoenix.cs.iupui.edu:4050/hh1

4.1.8. Model: Math domain's matrix operations model.

4.2. Functional Attributes:

4.2.1. Function Description: The method matrixMultiply takes two matrices as input parameters and returns the product of the two matrices.

4.2.2. Algorithm: Simple matrix multiplication algorithm.

4.2.3. Complexity: O (n3).

4.2.4. Syntactic Contract: double[ ][ ] matrixMultiply(in double[ ][ ] matrix1, in double[ ][ ] matrix2)

4.2.5. Technology: CORBA

4.2.6. Preconditions: if size(matrix1)=mxn and size(matrix2)=pxq then, m=p and n=q

4.2.7. Postconditions: if size(matrix1)=mxn and size(matrix2)=pxq then, size(matrix1*matrix2)= mxq

4.2.8. Invariant: matrix1,matrix2

4.2.9. Expected resources: none

4.2.10. Design Patterns: none

4.2.11. Known Usage: graphics

4.2.12. Alias: matrix product

5. Cooperation Attributes:

5.1. Preprocessing Collaborators: none

5.2. Postprocessing Collaborators: none

6. Auxiliary Attributes:

6.1. Mobility: no

6.2. Security: L1

6.3. Fault Tolerance: L1

7. QoS Metrics:

7.1. TURN-AROUND-TIME:

7.1.1. Effect of environment:

7.1.1.1. CPU Speed:

Table 5.1. CPU Speed vs. Turn-around-time

CPU Speed in MHZ@ Memory=128MB, Priority=0	Turn-around-time in milliseconds
100	91
200	74
300	50
400	18
500	15
600	12
700	11
800	9
900	8
1000	7

7.1.1.2. Memory:

Table 5.2. Memory vs. Turn-around-time

Memory in MB @ CPU Speed= 100MHZ, Priority=0	Turn-around-time in milliseconds
32	99
64	98
96	94
128	89
160	88
192	85
224	84
256	83
288	82
320	81

7.1.1.3. CPU speed and Memory:

Table 5.3. CPU speed, Memory vs. Turn-around-time

CPU speed in MHZ Memory in MB @ Priority=0	Turn-around-time in milliseconds
100MHZ, 32MB	99
200MHZ, 64MB	68
300MHZ, 96MB	48
400MHZ, 128MB	20
500MHZ, 160MB	14
600MHZ, 192MB	11
700MHZ, 224MB	10
800MHZ, 256MB	8
900MHZ, 288MB	7
1000MHZ, 320MB	6

7.1.1.4. Priority:

Table 5.4. Priority vs. Turn-around-time

Priority @ CPU Speed=100MHZ Memory=120MB	Turn-around-time in milliseconds
19	91
15	91
10	90
5	89
0	89
-5	88
-10	87
-15	85
-20	84

7.1.2. Effect of usage patterns:

7.1.2.1. Number of users:

Table 5.5. Number of users vs. Turn-around-time

Number of users @ Delay between requests=400 milliseconds	Turn-around-time in milliseconds
1	19
10	68
20	132
30	365
40	581
50	857
60	1361
70	1736
80	2174
90	2372
100	2792

7.1.2.2. Rate of requests:

Table 5.6. Delay between requests vs. Turn-around-time

Delay between requests @ Number of users=10	Turn-around-time in milliseconds
0	252
100	166
200	136
300	97
400	69
500	68
600	65
700	53
800	47
900	44
1000	41

7.1.2.3. Maximum delay between requests for Uniform distribution

Table 5.7. Maximum delay between uniformly distributed requests vs. Turn-around-time

Deviation of delay between requests in milliseconds@ Number of users=10, number of requests per user=50	Turn-around-time in milliseconds
0	250
100	215
200	163
300	134
400	93
500	78
600	68
700	65
800	57
900	55
1000	52

7.1.2.4. Deviation of delay between requests for Gaussian distribution

Table 5.8. Maximum delay between requests for Gaussian distribution of requests vs. Turn-around-time

Deviation of delay between requests in milliseconds@ Number of users=10, number of requests per user=50	Turn-around-time in milliseconds
0	250
100	175
200	139
300	99
400	86
500	72
600	68
700	57
800	54
900	48
1000	47

7.2. THROUGHPUT:

7.2.1. Effect of environment:

7.2.1.1. CPU Speed:

Table 5.9. CPU speed vs. Throughput

CPU Speed in MHZ @ Memory=128MB, Priority=0	Throughput in results/second
100	11
200	13.51
300	20
400	50.05
500	66.67
600	83.3
700	90.9
800	111.11
900	125
1000	142.86

7.2.1.2. Memory:

Table 5.10. Memory vs. Throughput

Memory in MB @ CPU Speed= 100MHZ, Priority=0	Throughput in milliseconds
32	10.1
64	10.2
96	10.64
128	11.05
160	11.36
192	11.76
224	11.9
256	12.05
288	12.2
320	12.35

7.2.1.3. CPU speed and Memory:

Table 5.11. CPU speed and Memory vs. Throughput

CPU speed in MHZ Memory in MB @ Priority=0	Throughput in results/second
100MHZ, 32MB	10.12
200MHZ, 64MB	14.71
300MHZ, 96MB	20.84
400MHZ, 128MB	50
500MHZ, 160MB	71.43
600MHZ, 192MB	90.1
700MHZ, 224MB	100
800MHZ, 256MB	125
900MHZ, 288MB	142.87
1000MHZ, 320MB	166.8

7.2.1.4. Priority:

Table 5.12. Priority vs. Throughput

Priority @ CPU Speed=100MHZ Memory=120MB	Throughput in results/second
19	10.78
15	10.8
10	11.9
5	11
0	11.04
-5	11.36
-10	11.5
-15	11.76
-20	11.9

7.2.2. Effect of usage patterns:

7.2.2.1. Number of users:

Table 5.13. Number of users vs. Throughput

Number of users @ Delay between requests=400 milliseconds	Throughput in results/second
1	52.63
10	14.5
20	7.57
30	2.78
40	1.72
50	1.17
60	0.73
70	0.58
80	0.46
90	0.42
100	0.36

7.2.2.2. Rate of requests:

Table 5.14. Delay between requests vs. Throughput

Delay between requests @ Number of users=10	Throughput in results/second
0	3.96
100	6.02
200	7.35
300	10.31
400	14.5
500	14.7
600	15.38
700	18.87
800	21.28
900	22.73
1000	24.39

7.2.2.3. Maximum delay between requests for Uniform distribution

Table 5.15. Maximum delay between requests for uniformly distributed requests vs. Throughput

Deviation of delay between requests in milliseconds@ Number of users=10	Throughput in results/second
0	3.95
100	4.65
200	6.13
300	7.46
400	10.75
500	12.82
600	14.7
700	15.38
800	17.54
900	18.18
1000	19.23

7.2.2.4. Deviation of delay between requests for Gaussian distribution

Table 5.16. Maximum Delay between requests for Gaussian distribution of requests vs. Throughput

Deviation of delay between requests in milliseconds@ Number of users=10	Throughput in results/second
0	3.96
100	5.71
200	7.19
300	10.1
400	11.63
500	13.89
600	14.7
700	17.54
800	18.52
900	20.83
1000	21.28

8. Heterogeneity Bridging rules: none
9. Interaction: none
10. Interaction: none
11. Event Grammar Rules: none
12. Deployment Rules: none
13. Configuration Knowledge:
    - 13.1. Illegal feature combinations: none
    - 13.2. Default settings: none
    - 13.3. Default dependencies: none
    - 13.4. Construction rules: none
    - 13.5. Optimizations: none
    - 13.6. Concrete component selection rules: none

The component is now ready for deployment. The component developer would now be able to deploy the component on a network to be located by the head-hunters. This marks the end of the component development phase and the beginning of the system development phase of the UniFrame approach.

6. CONCLUSION

This thesis presented the UQOS framework which is an implementation of the QoS aspect of the UniFrame project. This chapter is the concluding chapter of the thesis. It presents an overview of the features of the UQOS framework in section 6.1, followed by the possible future enhancements to the UQOS framework in section 6.2 and concludes with a summation in section 6.3.

6.1. Features of the UQOS framework

The features of the UQOS framework are as follows:

- It provides a framework to objectively quantify the Quality of Service of software components. This framework is implemented in four major parts as presented below.

- It provides a Quality of Service Catalog to standardize the notion of quality of software components and to act as a reference guide for software component developers (producers) and system integrators (consumers).

- It provides a standard approach to incorporate the effect of environment on QoS, into the component development process.

- It provides a standard approach to incorporate the effect of usage patterns on QoS, into the component development process.

- It provides a means to specify the QoS of software components by adopting the Component Quality Markup Language (CQML).

6.2. Future Work

Some of the possible future enhancements to the UQOS framework are as follows:

- Further enhancement of the QoS Catalog by inclusion of more QoS parameters.
- Incorporation of QoS-based composition and decomposition rules at the system-level into the UQOS framework.
- Incorporation of mechanisms for dynamic QoS-based adaptation of a component/system. This would also involve mechanisms for dynamic QoS-based negotiation and verification.
- Incorporation of a compensation scheme for the negotiation of QoS.
- Formal specification of a component and its QoS using Two-Level Grammar.
- The validation and assurance of QoS, based on the concept of Event Grammars.

6.3. Summation

This thesis presented a QoS framework, called the UniFrame Quality of Service (UQOS) framework as a part of the UniFrame Project, to address the issues related to standardization of the QoS of software components, the effect of environment and the effect of usage patterns on the QoS of software components, and the specification of the QoS of software components. The UQOS framework has been implemented in four

major parts, namely, the QoS Catalog for software components, the approach for accounting for the effects of environment on the QoS of software components, the approach for accounting for the effects of usage patterns on the QoS of software components and the specification of the QoS of software components. The QoS Catalog is intended to act as a tool for standardizing the notion of the Quality of software components. The catalog contains detailed descriptions about QoS parameters of software components, including the metrics, the evaluation methodologies, the factors influencing these parameters and the interrelationships among these parameters. The studies related to the effects of the environment and the effects of usage patterns on the QoS of components propose standard approaches to account for the effects of the environment and the effects of usage patterns on the QoS of the software components. They consist of an empirical validation of the QoS of software components under diverse environmental conditions and usage patterns, and specification of the resulting QoS values in the component interface. The thesis also presented a case-study to illustrate the application of the UniFrame approach and the UQOS framework in a real-world scenario. It is believed that the UQOS framework together with the UniFrame Approach would provide a promising solution to the automatic generation of heterogeneous DCS with QoS guarantees.

LIST OF REFERENCES

[AAG01] Aagedal J. Ø., "Quality of Service Support in Development of Distributed Systems", PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[ACE02] Schmidt D., "The ACE overview", http://www.cs.wustl.edu/~schmidt/ACE-overview.html, 2002.

[AUG95] Auguston M., "Program Behavior Model based on Event Grammar and its application for debugging automation", Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, 1995.

[AUG97] Auguston M., Gates A., Lujan M., "Defining a program behavior model for dynamic analyzers", Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering", pages 257-262, 1997.

[BAR00] Barrett R. B., "Object-Oriented Natural Language Requirements Specification", In Proceedings of ACSC 2000, The 23rd Australasian Computer Science Conference, January 31-February 4, 2000, Canberra, Australia, pages 24-30, January 2000.

[BBN01] BBN Corporation, Quality Objects Project Url: http://www.dist-systems.bbn .com/ tech/QuO, 2001.

[BER01] Berzins V., Shing M., Auguston M., Bryant B., Kin B., "DCAPS-Architecture for Distributed Computer Aided Prototyping System", Proceedings of RSP 2001, the 12th international workshop on rapid system prototyping , 2001.

[BEU99] Beugnard A., Jézéquel J., Plouzeau N., Watkins D., Making components contract aware, *IEEE Computer*, 13(7), July 1999, pages 38-45.

[BLA98] Blake S., "An Architecture for Differentiated Services", RFC 2475, December 1998.

[BLA01] Blair G.S., Coulson G., "OpenORB", IEEE Distributed Systems Online, Vol. 2, No. 6, 2001.

[BRA01] Brahnmath G., Raje R., Olson A., Sun C., "Quality of Service Catalog for Software Components", Technical Report (TR-CIS-0219-01), Department of Computer and Information Science, Indiana University Purdue University Indianapolis, 2001.

[BRU98] Bruno J., Gabber E., Özden B., and Silberschatz A., "The Eclipse Operating System: Providing Quality of Service via Reservation Domains", Proceedings of the USENIX 1998 Annual Technical Conference, New Orleans, Louisiana, June 1998.

[BUR02] Carol C. Burt, Barrett R. Bryant, Rajeev R. Raje, Andrew Olson. Mikhail Auguston, "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," pages 212-223, Proceedings of the 6th IEEE International Enterprise Distributed Object Computing Conference, Lausanne, Switzerland, September 2002.

[CAM96] Campbell A., "A Quality of Service Architecture", PhD Thesis, Computing Department, Lancaster University, 1996.

[CAZ99] Cazzola W. et al., "Rule-based Strategic Reflection: Observing and Modifying Behavior at the Architectural Level", Proc. 14th IEEE Int'l Conf. Automated Software Engineering (ASE 99), IEEE Press, Piscataway, N.J., Oct. 1999, pages 263-266.

[CHE95] Cheaito R., Frappier M., Matwin S., Mili A., Crabtree D., "Defining and Measuring Maintainability", Technical Report, Dept. of Computer Science, University of Ottawa, March 1995.

[CHU01] Chung L., Subramanian N., "Process-Oriented Metrics for Software Architecture Adaptability", Proceedings of International Symposium On Requirements Engg., IEEE Computer Press, Aug - Sep. 2001, pp. 310-311.

[CID02] OMG CORBA IDL Specification, http://www.omg.org/gettingstarted/omg_idl. htm, 2002.

[CLA94] Clark D., Braden R., and Shenker S., "Integrated Services in the Internet Architecture: an Overview", Internet RFC 1633, June 1994.

[DTF00] OMG Special Interest Groups, Task Forces, and the Architecture Board, http://cgi.omg.org/techprocess/sigs.html, 2000.

[FER98] Ferguson P., Huston G., "Delivering QoS on the Internet and in Corporate Networks", John Wiley & Sons, January 1998, ISBN 0-471-24358-2.

[FLO96] Florissi, P., "QuAL: Quality Assurance Language", Ph.D. Thesis, Columbia University, 1996.

[FRA94] Frappier M., Matwin S., Mili A., "Maintainability: Factors and Criteria", Software Metrics Study: Technical Memorandum 1, Canadian Space Agency, March 1994.

[FRO98] Frolund S., Koistinen J., "Quality of Service specification in distributed object systems", Distributed System Engineering Journal, Vol.5, Issue 4, December 1998.

[GAM95] Gamma E., Helm R., Johnson R., Vlissides J., “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley Publication Company, 1995.

[GAR97] Garrett M. W., "A Service Architecture for ATM: From Applications to Scheduling", IEEE Network Magazine, pages 6-14, May 1996.

[HEI99] Heinanen J., et al., “Assured Forwarding PHB Group”, RFC 2597, June 1999

[HUA02] Huang Z.,Raje R., Olson A., Bryant B., Auguston M., Burt C., Sun C., "System-Level Generative Programming of Unified Approach Based on UMM for the Integration of Distributed Software Components", Proceedings of the IEEE Fifth International Conference on Algorithms and Architectures for Parallel Processing, Beijing, China, October 2002.

[ISO, 1986], Quality Vocabulary, ISO, Report: ISO 8402, pp. 8.

[ISO99] ISO/IEC JTC1/SC7, “Information Technology - Software product quality: Quality model,” ISO/IEC, 9126, 1999.

[JME02] Apache JMeter, http://jakarta.apache.org/jmeter/, 2002.

[KAS02] Kasten E., McKinley P.K., Sadjadi S., and Stirewalt R., ``Separating introspection and intercession in metamorphic distributed systems," Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing, Vienna, Austria, July 2002.

[KON00] Kon F., Mickunas M., Nahrstedt K., and Ballesteros F., “2K: A Distributed Operating System for Dynamic Heterogeneous Environments”, 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh. August 1-4, 2000.

[KON01] Kon F., Yamane T., Hess C., Campbell R., Dennis M., “Dynamic Resource Management and Automatic Configuration of Distributed Component Systems”, Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas, January, 2001.

[KUR01] Kurose J., Ross K., “Computer Networking: A Top-Down Approach”, Addison-Wesley, 2001, ISBN 0-201-47711-4.

[LES96] Leslie I., McAuley D., Black R., Roscoe T., Barham P., Evers D., Fairbairns R. and Hyden E., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", IEEE Journal on Selected Areas in Communications, Vol. 14, No. 7, September 1996, pages 1280-1331.

[LOY98] Loyall J., Bakken D., Schantz R., Zinky J., “QoS Aspect Languages and Their Runtime Integration”, Lecture Notes in Computer Science, Vol. 1511, Springer-Verlag, Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98), pages 28-30, May 1998, Pittsburgh, Pennsylvania.

[MAY02] Mayo S., “Web Services: How Will Professional Services Firms Compete for This Multibillion-Dollar Opportunity”, IDC, March 2002.

[MCK99] McKinley P.K., Malenfant A.M., Arango J.M., “Pavilion: A distributed middleware framework for collaborative web-based applications”, Proceedings of the ACM SIGGROUP conference on supporting group work, pages 179-188, 1999.

[MCK01] McKinley P.K., Padmanabhan U. I., and Ancha N., ``Experiments in composing proxy audio services for mobile users", Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, pages 99-120, November 2001.

[MIC02] Microsoft Online Clipart Gallery, http://dgl.microsoft.com/default.asp, 2002.

[MID02] Microsoft IDL reference, http://msdn.microsoft.com /library/default.asp?url =/library/en-us/midl/midl/midl_language_reference.asp, 2002.

[MOO97] Mooney J., “Bringing Portability to the Software Process”, Technical Report, TR-97-1, Dept. of Statistics and Computer Science, West Virginia University, 1997.

[MOO93] Mooney J., “Issues in the Specification and Measurement of Software Portability”, Technical Report, TR-93-6, Dept. of Statistics and Computer Science, West Virginia University, 1993.

[OMG99] *CORBA Components - Volume 1*, Object Management Group, Report: orbos/99-07-01, 1999.

[OMG02] Object Management Group. 2002. “UML™ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms”. Request for Proposal, OMG document ad/02-01-07, Framington, MA.

[PIK95] Pike R., et al, “Plan 9 from Bell Labs”, Computing Systems, The Journal of the USENIX Association, pages 221-254, summer 1995.

[RAJ01] Raje R., Auguston M., Bryant B., Olson A., Burt C. “A Quality of Service – based framework for creating Distributed Heterogeneous Software Components”, Submitted to Informatica, 2001.

[RAJ00] Raje R., "UMM: Unified Meta-object Model for Open Distributed Systems", Proceedings of the fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP'2000).

[RAJM01] Raje R., Auguston M., Bryant B., Olson A., Burt C., "A Unified Approach for the Integration of Distributed Heterogeneous Software Components", Proceedings of the 2001 Monterey Workshop, Monterey, California, 2001.

[SCH98] Schmidt D., Levine D., Mungee S., “The Design of the TAO Real-Time Object Request Broker”, Computer Communications Journal, Volume 21, No 4, April, 1998.

[SHE97] Shenker, S., and J. Wroclawski, "General Characterization Parameters for Integrated Service Network Elements", RFC 2215, September 1997.

[SIM02] Magnusson P., Christensson M., Eskilson J., Forsgren D., Simics: A Full System Simulation Platform, IEEE Computer, February 2002, pages 50-58.

[SIR02] Siram N., “An Architecture for the UniFrame Resource Discovery Service”, Masters Thesis, Purdue University, May 2002.

[SUB01] Subramanian N., Chung L., "Software Architecture Adaptability - An NFR Approach", Proceedings of International Workshop on Principles of Software Evolution, Vienna, September 2001.

[SUN02] Sun C., Raje R., Olson A., Bryant B., Auguston M., Burt C., Huang Z., "Composition and Decomposition of Quality of Service Parameters in Distributed Component-Based Systems", Proceedings of the IEEE Fifth International Conference on Algorithms and Architectures for Parallel Processing, Beijing, China, October 2002

[SZY99] Szyperski C., “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, ISBN 0-201-17888-5, 1999, pg 34.

[TOM02] Apache Tomcat, http://jakarta.apache.org/tomcat/index.html, 2002.

[UML RTF, 1999] *OMG UML v1.3*, Object Management Group, Report: ad/99-06-08, 1999.

[VAN65] Van A., “Orthogonal Design and Description of a Formal Language”, Technical report, Mathematisch Centrum, Amsterdam, 1965.

[VOA95] Voas J., Software Testability Measurement for Assertion Injection and Fault Localization, Proceedings of 2nd Int'l. Workshop on Automated and Algorithmic Debugging (AADEBUG'95), St. Malo, France, May 1995.

[VOA96] Voas J., Ghosh A., McGraw G., Charron F., and Miller K, Defining an adaptive software security metric from a dynamic software failure tolerance measure, Proceedings of the 1l th Annual Conference on Computer Assurance, June 1996, 250-263.

[VOA98] Voas J., An Approach to Certifying Off-the-Shelf Software Components, IEEE Computer, June, 1998.

[VOA00] Voas J., Payne J., Dependability Certification of Software Components, Journal of Components and Software, 2000.

[WAN00] Wang P., Yemini Y., Florissi D. and Florissi P., "QoSME: Toward QoS Management and Guarantees", World Computer Congress - International Conference on Communication Technologies, Beijing, China, August 2000.

[WES02] The Web Services Community Portal, http://www.webServices.org, 2002.

[ZHA96] Zhang L., Berson S., Herzog S. and Jamin S. "Resource Reservation Protocol (RSVP) - Version 1 Functional Specification", August, 1996.